
UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

---

Corso di Laurea Magistrale in Informatica

# Progettazione e Sviluppo di una Pipeline End-to-End di Object Detection e Offuscamento per Sistemi Intelligenti di Monitoraggio Cantieri

*Relatore:*

**Prof. Luca Bedogni**

*Tesi di Laurea di:*

**Marco Michellini**

---

Anno Accademico 2025-2026

**Sommario**

L'automazione del monitoraggio nei cantieri edili tramite sistemi di ripresa *timelapse* ad altissima risoluzione (4K) impone sfide complesse, sia dal punto di vista computazionale che normativo. Il presente lavoro di tesi, sviluppato in collaborazione con l'azienda TimeLapse-Lab, propone la progettazione, l'ottimizzazione e la messa in produzione di un'infrastruttura architetturale *end-to-end* per l'Object Detection e l'anonimizzazione dei dati visivi, nel rigoroso rispetto dei principi di *Privacy-by-Design* stabiliti dal GDPR.

La ricerca si è concentrata sull'implementazione di modelli di Deep Learning allo stato dell'arte, appartenenti alla famiglia YOLO (in particolare YOLOv11), per il rilevamento multiclasse di Dispositivi di Protezione Individuale (elmetti), targhe automobilistiche, pedoni e loghi aziendali. Per arginare la fisiologica perdita di dettaglio spaziale causata dal ridimensionamento nativo dei tensori sulle immagini 4K, è stata implementata un'avanzata pipeline di *Data Engineering*. Questa ha previsto l'applicazione di tecniche di *tiling* geometrico e l'integrazione in inferenza della libreria SAHI (*Slicing Aided Hyper Inference*), superando criticità algoritmiche legate al *Data Leakage* e all'allocazione di memoria contigua su GPU.

Per supportare il *deployment* industriale in totale scalabilità, è stato inoltre ingegnerizzato un *Inference Server* ibrido basato su protocolli HTTP e MQTT. Il disaccoppiamento asincrono tra la ricezione del dato e l'elaborazione su *worker* isolati ha ottimizzato la latenza di sistema, rimuovendo i colli di bottiglia derivanti dal trasferimento di carichi pesanti. I risultati sperimentali certificano metriche eccellenti (mAP e F1-Score) sui dataset proprietari, validando l'efficacia del sistema nel combinare il rilevamento ad alta precisione su piccola scala con algoritmi di offuscamento affidabili e irreversibili.

# Indice

# Capitolo 1

# Introduzione

## 1.1 Il contesto aziendale: TimeLapseLab

Il presente lavoro di tesi è il risultato dell'attività di tirocinio curricolare svolta presso TimeLapseLab, realtà innovativa con sede a Mantova specializzata nel monitoraggio intelligente di cantieri edili e siti industriali attraverso l'utilizzo di sistemi di ripresa a intervalli temporali (timelapse).

L'azienda opera in un dominio estremamente sfidante: la gestione e l'analisi automatizzata di ingenti volumi di dati visivi ad altissima risoluzione (4K). Le telecamere installate nei siti monitorati producono flussi continui di immagini che, una volta trasmesse ai server aziendali, devono essere non solo archiviate, ma interpretate per fornire valore aggiunto ai clienti.


Tra i servizi offerti rientrano il monitoraggio dell'avanzamento dei lavori, la verifica della sicurezza sul cantiere e la produzione di materiale per il marketing. In questo scenario, l'intervento umano per la revisione manuale di ogni singolo frame risulta impraticabile in termini di tempi e costi, rendendo necessaria l'implementazione di soluzioni scalabili basate sulla Computer Vision e sul Deep Learning.

## 1.2 Motivazioni e Problematiche

L'automazione del monitoraggio in ambienti aperti e non controllati, come i cantieri edili, presenta sfide tecniche e legali di notevole rilievo che hanno guidato lo sviluppo di questo progetto:

- **Sicurezza sul Lavoro (Safety):** Il monitoraggio dell'utilizzo dei Dispositivi di Protezione Individuale (DPI) è un requisito critico per la sicurezza nei cantieri. Rilevare automaticamente la presenza o, più criticamente, l'assenza dell'elmetto protettivo (*hardhat*) sul personale operativo permette di prevenire infortuni e garantire il rispetto delle rigorose normative di sicurezza vigenti.

- **Privacy e Compliance Normativa (GDPR):** Poiché i dispositivi di acquisizione riprendono continuamente aree in cui operano lavoratori e transitano mezzi privati e commerciali, è fondamentale garantire l'anonimizzazione dei dati sensibili fin dalle prime fasi di elaborazione. Questo processo, noto come *Privacy-by-Design*, include l'offuscamento dei volti e delle sagome dei lavoratori, delle targhe dei veicoli e delle scritte pubblicitarie (loghi) sui furgoni che potrebbero ricondurre all'identità di aziende terze non autorizzate.

- **Alta Risoluzione vs Vincoli Computazionali:** L'elaborazione nativa di immagini in formato 4K (3840×2160 pixel) da parte di Reti Neurali Convoluzionali richiede enormi quantitativi di memoria video (VRAM). D'altro canto, un ridimensionamento drastico comporterebbe la perdita di dettagli fondamentali, come targhe di veicoli situati in lontananza. È stato quindi necessario studiare tecniche di *tiling* avanzate e inferenza ottimizzata per bilanciare l'accuratezza visiva con i colli di bottiglia computazionali.

- **Variabilità Ambientale (Domain Gap):** Il sistema deve operare nello stato che in letteratura è definito come *in the wild*. L'architettura deve dimostrarsi robusta rispetto a forti cambiamenti di illuminazione, condizioni meteorologiche avverse, ombre proiettate, occlusioni parziali (ad esempio dovute a impalcature o reti da cantiere) e angolazioni di ripresa atipiche, spesso di tipo *top-down*.

## 1.3 Obiettivi della Tesi

L'obiettivo principale di questo progetto è lo sviluppo, l'ottimizzazione e la messa in produzione di una pipeline architetturale di Object Detection end-to-end, capace di elaborare i flussi di immagini provenienti dalle telecamere di TimeLapseLab garantendo massima accuratezza nel rilevamento e totale protezione della privacy.

Nello specifico, il lavoro di ricerca e sviluppo si è articolato nei seguenti punti chiave:

- **Sviluppo e Ottimizzazione di Modelli:** Ricerca, addestramento e *fine-tuning* di reti neurali allo stato dell'arte basate su YOLOv11 per il rilevamento delle seguenti classi: DPI (caschetti), targhe automobilistiche, sagome umane (pedoni) e grafiche pubblicitarie sui mezzi di trasporto.

- **Ingegneria del Dato (Data Engineering):** Implementazione di workflow locali e in ambiente Cloud per la preparazione del dato. Questo include il preprocessing geometrico delle immagini 4K, lo sviluppo di logiche matematiche per il *tiling* intelligente, la prevenzione del *data leakage* e il bilanciamento di dataset ibridi composti da immagini *open-source* e dati proprietari del dominio target.
- **Architettura Software per l'Inferenza:** Progettazione e implementazione di un *Inference Server* ibrido, disaccoppiato e asincrono basato su protocolli HTTP e Message Brokers (MQTT). Il sistema è stato concepito per gestire l'elevato carico computazionale derivante dai flussi 4K in modo distribuito, prevenendo la saturazione del server web aziendale.
- **Privacy-by-Design:** Sviluppo e integrazione nativa di moduli di *obfuscation* algoritmica applicati sulle coordinate (*bounding box*) predette dai modelli di rilevamento, al fine di garantire l'anonimizzazione irreversibile delle entità sensibili prima del salvataggio definitivo dell'immagine.

## 1.4 Struttura dell'Elaborato

Il presente documento è organizzato come segue:

- **Capitolo 2:** Viene presentato lo Stato dell'Arte, introducendo i concetti fondamentali della *Computer Vision*, l'evoluzione della famiglia di modelli YOLO (con focus sulla versione v11), e le tecnologie architetturali per la gestione asincrona come il protocollo MQTT.
- **Capitolo 3:** Si approfondisce la fase critica dell'Ingegneria dei Dati, descrivendo matematicamente le tecniche di *tiling* per immagini ad alta risoluzione, la gestione del bilanciamento del *background* e la creazione di dataset ibridi per la mitigazione del *Domain Gap*.
- **Capitolo 4:** Viene descritto il processo di training, documentando gli esperimenti condotti su Cloud GPU, la validazione dei modelli specifici e l'analisi rigorosa delle metriche valutative (Precision, Recall, mAP).
- **Capitolo 5:** Si illustra la progettazione software del *Backend*, dettagliando il funzionamento del *Worker* di inferenza indipendente, il passaggio dei metadati tramite MQTT e l'integrazione degli script di *obfuscation*.
- **Capitolo 6:** Vengono riportati e analizzati criticamente i risultati sperimentali sul *test set* reale, unitamente a un'analisi delle prestazioni latenziali (*stress test*) del sistema integrato. Vengono inoltre raccolte le conclusioni sul lavoro svolto, evidenziandone

i risultati raggiunti e delineando le possibili traiettorie di sviluppo futuro in ottica di ottimizzazione hardware e deployment Edge.

# Capitolo 2

# Stato dell'Arte e Tecnologie

## 2.1 Fondamenti di Object Detection moderna

L'Object Detection rappresenta uno dei task più complessi della Computer Vision [8], in quanto richiede la risoluzione simultanea di due problemi: la classificazione (determinare *cosa* è presente) e la localizzazione (determinare *dove* si trova tramite bounding box).

Nel panorama moderno le architetture si dividono principalmente in due categorie:

- **Two-stage Detectors (es. Faster R-CNN):** Utilizzano una *Region Proposal Network* (RPN) per identificare zone candidate e successivamente una rete di classificazione. Pur essendo estremamente precisi, presentano latenze elevate, rendendoli spesso inadatti ad applicazioni real-time.
- **One-stage Detectors (es. famiglia YOLO):** Trattano la detection come un unico problema di regressione, mappando i pixel dell'immagine direttamente sulle coordinate delle bounding box e sulle probabilità di classe in un unico passaggio (feed-forward). Questa categoria è quella che meglio si presta al deployment industriale in pipeline ad alte prestazioni.

## 2.2 Evoluzione della famiglia YOLO: da v1 a v11

L'architettura **YOLO (You Only Look Once)** ha rivoluzionato il settore introducendo il concetto di grid-based detection. Dalla sua prima versione (2016, [1]) all'attuale **YOLOv11** [2], l'evoluzione è stata guidata dalla ricerca di un bilanciamento ottimale tra efficienza computazionale (FLOPS) e accuratezza (mAP).

Le innovazioni chiave integrate nelle ultime iterazioni includono:

- **Backbone e Neck avanzati:** L'introduzione di blocchi CSP (Cross Stage Partial) e moduli C3/C2f ha permesso una propagazione del gradiente più efficace e una riduzione dei parametri.

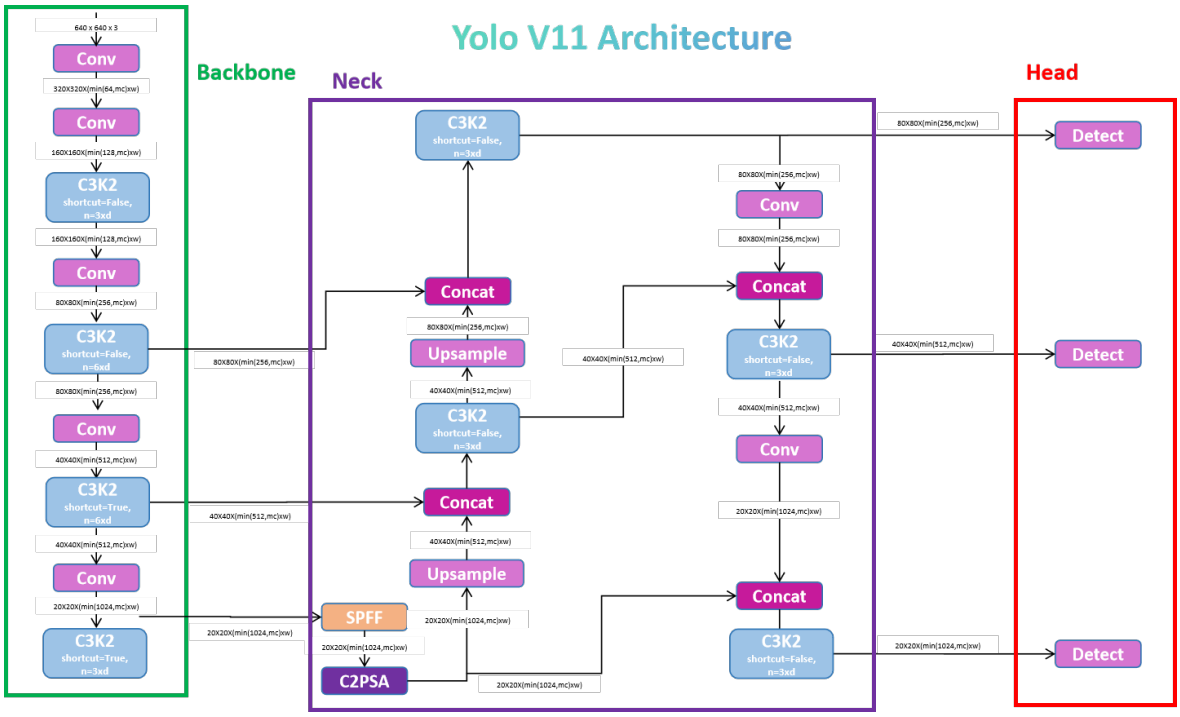



Figura 2.1: Diagramma a blocchi che illustra l'architettura di YOLOv11, nei tre blocchi principali di Backbone, Neck ed Head[3].

- **Anchor-free Detection:** A differenza delle prime versioni che utilizzavano *anchor boxes* predefinite, i modelli recenti predicono direttamente il centro dell'oggetto e la distanza dai bordi, migliorando la capacità di generalizzazione su oggetti con *aspect ratio* estremi.
- **Decoupled Heads:** La separazione dei rami di classificazione e regressione permette alle due funzioni di convergere più velocemente, riducendo i conflitti durante l'addestramento.
- **YOLOv11 Specifics:** L'ultima versione ottimizza ulteriormente i blocchi C3k2 e integra meccanismi di attenzione per migliorare il rilevamento in scenari a bassa densità di feature, aspetto critico per il riconoscimento di piccoli dettagli in immagini ricche di informazioni.

## 2.3 Problematica della Small Object Detection e SAHI

Uno dei limiti intrinseci delle reti neurali convoluzionali è la perdita di risoluzione spaziale dovuta alle operazioni di pooling e allo stride dei layer convoluzionali. Quando un'immagine 4K viene ridimensionata a 640x640 per essere processata da YOLO, un oggetto piccolo (come un elmetto in lontananza) può ridursi a pochi pixel, rendendo impossibile l'estrazione di feature discriminanti e di conseguenza il rilevamento risulta molto complesso.

Per ovviare a ciò, si utilizza la tecnica della **Slicing Aided Hyper Inference (SAHI [4])**. SAHI non è un nuovo modello, ma una pipeline che suddivide l'immagine originale in *fette* (tile) sovrapposte, esegue l'inferenza su ciascuna fetta e infine unifica i risultati tramite algoritmi di **Non-Maximum Suppression (NMS)** o **Non-Maximum Merging**.

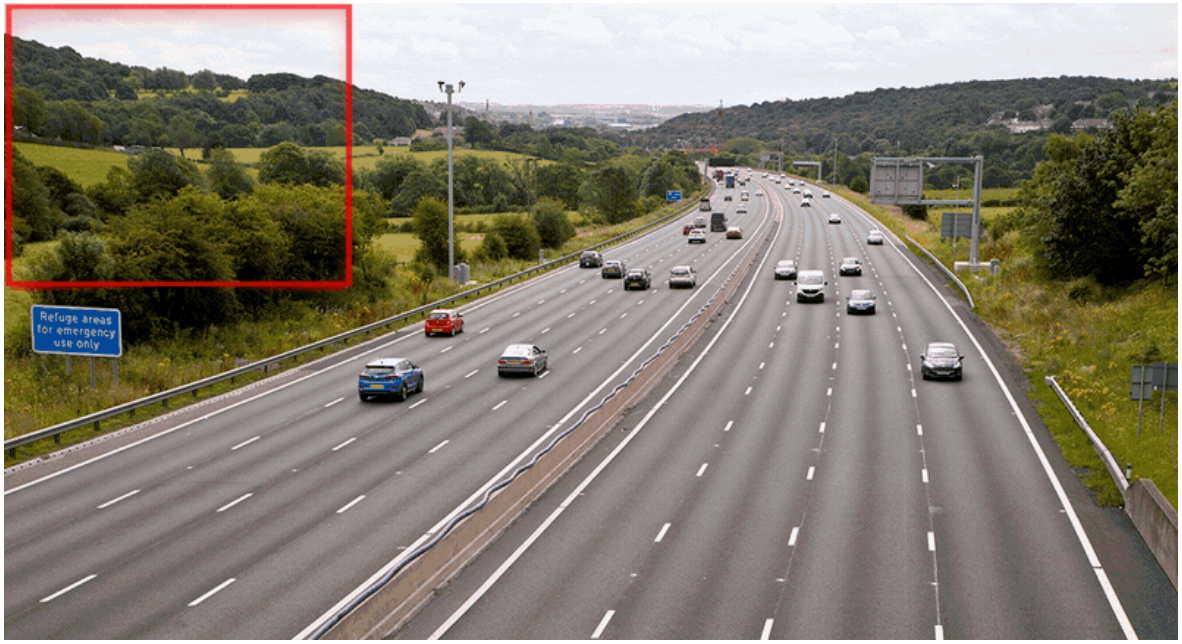



Figura 2.2: Immagine di esempio per SAHI[5]. Il rettangolo rosso scorre tutta l'immagine, creando porzioni di essa.

## 2.4 Il protocollo MQTT nell'architettura MLOPS

In un ambiente di produzione, l'inferenza Deep Learning è un'operazione *compute-intensive* che non dovrebbe mai bloccare il flusso principale dell'applicazione (es. la ricezione dell'immagine dal server).

Il protocollo **MQTT [6] (Message Queuing Telemetry Transport)**, basato sul paradigma Publish/Subscribe, è stato scelto per gestire questa comunicazione asincrona. A differenza del modello Request/Response del protocollo HTTP, MQTT offre:

- **Disaccoppiamento temporale e spaziale:** Il produttore del dato (l'Executor) e il consumatore (l'Inference Server) non devono conoscersi né essere attivi contemporaneamente.
- **Qualità del Servizio (QoS):** La gestione dei livelli di QoS garantisce che i messaggi di inferenza (job) vengano consegnati correttamente anche in presenza di instabilità della rete.
- **Efficienza:** L'overhead del protocollo è minimo, ideale per sistemi distribuiti che gestiscono frequenze di campionamento elevate.

## 2.5 Data Augmentation e Test-Time Augmentation (TTA)

Nel contesto del Deep Learning per la Computer Vision, la scarsità o la bassa varianza dei dati può portare al fenomeno dell'*overfitting*, dove il modello memorizza le caratteristiche del dataset di addestramento perdendo capacità di generalizzazione. La **Data Augmentation** interviene in fase di training applicando trasformazioni stocastiche (rotazioni, variazioni di luminosità, *Mosaic Augmentation*) per espandere artificialmente il dataset.

Una tecnica avanzata utilizzata durante lo sviluppo dei modelli è la **Test-Time Augmentation (TTA)**. A differenza dell'augmentation tradizionale, la TTA agisce in fase di inferenza: l'immagine di input viene moltiplicata in diverse versioni (es. specchiata o scalata) e sottoposta a molteplici passaggi di detection. I risultati vengono poi aggregati tramite tecniche di *ensemble*, permettendo al sistema di recuperare oggetti che in un singolo passaggio standard avrebbero ottenuto punteggi di confidenza insufficienti.

## 2.6 Metriche di Valutazione per l'Object Detection

Per valutare oggettivamente le performance dei modelli addestrati, è necessario definire metriche che tengano conto sia della precisione della localizzazione che della correttezza della classificazione. Le metriche utilizzate per valutare la qualità dei modelli ottenuti sono le seguenti:

- **Intersection over Union (IoU):** Misura la sovrapposizione tra la *Bounding Box* predetta ($B_p$) e quella reale (*Ground Truth*, $B_g$). È definita come il rapporto tra l'area della loro intersezione e l'area della loro unione:

$$IoU = \frac{Area(B_p \cap B_g)}{Area(B_p \cup B_g)} \tag{2.1}$$

  Una predizione è considerata un **True Positive (TP)** se l'IoU supera una soglia prestabilita (solitamente 0.5). In caso contrario, o se la classe è errata, si parla di **False Positive (FP)**. Un oggetto reale non rilevato è un **False Negative (FN)**. Il concetto è visualizzato nella figura 2.3.

- **Precision e Recall:** La *Precision* indica la capacità del modello di non classificare come positivo un campione negativo, mentre la *Recall* indica la capacità di trovare tutti i campioni positivi.

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \tag{2.2}$$

- **Mean Average Precision (mAP):** Rappresenta la metrica riassuntiva principale. Si ottiene calcolando l'area sotto la curva Precision-Recall (Average Precision) per ogni

classe e mediando i risultati. Solitamente si valuta il **mAP@.5** (soglia IoU fissa a 0.5) o il **mAP@.5:.95** (media su diverse soglie di IoU per premiare la precisione geometrica).

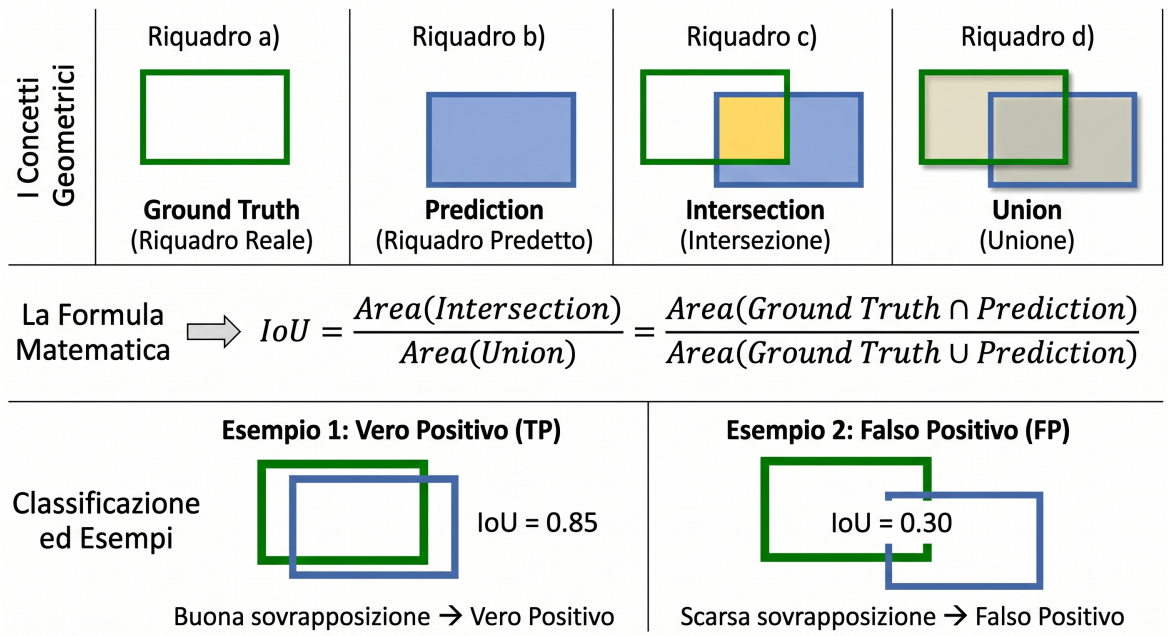



Figura 2.3: Rappresentazione del concetto di IoU, Vero Positivo (TP) e Falso Positivo (FP).

## 2.7 Privacy-by-Design e GDPR

L'integrazione di sistemi di videosorveglianza intelligente in contesti lavorativi richiede la conformità al Regolamento Generale sulla Protezione dei Dati (GDPR [7]). Il principio di *Privacy-by-Design* impone che la protezione dei dati sia integrata nel sistema sin dalla fase di progettazione.

Tecnicamente, questo si traduce nell'applicazione di filtri di *offuscamento (blurring o pixelation)*. La sfida ingegneristica consiste nel trasformare le coordinate continue fornite dal modello di detection in matrici di convoluzione (filtri Gaussiani) applicate selettivamente sui tensori dell'immagine, garantendo che l'anonimizzazione sia irreversibile prima del salvataggio dei dati su memoria persistente.

# Capitolo 3

# Ingegneria dei Dati e Preparazione del Dataset

## 3.1 Introduzione e Analisi del Dominio Visivo


Figura 3.1: Esempio di immagine scattata in un cantiere dalla telecamera di TimeLapseLab

Rielaborare il dato di input per ottenere qualità e coerenza rappresenta un passaggio fondamentale per la creazione di un sistema di Deep Learning efficace. Nel contesto aziendale di TimeLapseLab, le immagini fornite dalle telecamere di cantiere si presentano in formato 4K (3840×2160 pixel, un esempio nella figura 3.1). Sebbene l'alta risoluzione sia un vantaggio per l'identificazione di dettagli con piccole dimensioni, rappresenta invece un ostacolo per le moderne architetture di Object Detection. Modelli come YOLOv11, infatti,

operano nativamente su tensori di dimensioni quadrate notevolmente inferiori (es. 640×640 o 800×800).

Un ridimensionamento (*resize*) diretto dell'immagine originale a 640×640 comporterebbe una distorsione dell'*aspect ratio* e una perdita notevole di informazioni, rendendo di fatto invisibili oggetti come targhe o elmetti situati in background. Per aggirare questo problema, è stata implementata una pipeline di preprocessing geometrico basata sul *tiling* (sezionamento).

## 3.2 Preprocessing Geometrico: La Strategia di Tiling

L'obiettivo del *tiling* è suddividere l'immagine 4K originale in una griglia di *patch* (fette) di dimensioni inferiori, in modo da preservare la qualità e il livello di dettaglio originale. Nel presente progetto, la dimensione ottimale della *patch* è stata empiricamente fissata a 1400×800 pixel, dimensione che garantisce un buon *trade-off* tra preservazione dei dettagli nella singola fetta e un numero ragionevole di fette, aspetto fondamentale che impatta il tempo di inferenza.

Tuttavia, l'implementazione di una finestra scorrevole (*sliding window*) su immagini con dimensioni non multipli esatti della patch ha fatto emergere due criticità algoritmiche significative, risolte tramite logiche custom.

### 3.2.1 Risoluzione delle Distorsioni di Bordo

Nella prima iterazione dell'algoritmo, i ritagli situati ai margini destro e inferiore dell'immagine originale risultavano di dimensioni ridotte (ad esempio, 240×800 pixel). Quando queste fette venivano passate a YOLO per l'addestramento, la rete forzava un ridimensionamento a 640×640, causando una compressione severa (distorsione prospettica) che degradava drasticamente la capacità del modello di apprendere le *feature* corrette.

Per ovviare a questo problema, è stata introdotta una logica di controllo di dimensioni. Lo script è stato modificato affinché, qualora la finestra scorrevole dovesse superare i limiti dell'immagine originale, le coordinate di taglio vengano forzatamente arretrate per coprire esattamente gli ultimi 1400 o 800 pixel disponibili, garantendo così che l'output finale sia *tassativamente* di 1400×800 pixel.

### 3.2.2 Ottimizzazione del Passo e Overlap

Un aspetto fondamentale che è stato preso in considerazione è quello dell'*overlap* (sovrapposizione) tra le fette, necessario nei casi dove l'area di interesse (es. persona) è posizionata tra due tile. L'introduzione di sovrapposizione permette alla fetta successiva di ricoprire l'area finale di quella precedente, evitando così il problema. Inoltre, tramite semplici calcoli matematici, il valore di *stride* (passo di scorrimento) è stato impostato a 1220×680, risultando

quindi perfettamente in 9 tile per immagine (3×3) e garantendo una sovrapposizione costante di 180×120 pixel, come rappresentato nella figura 3.2.


Figura 3.2: Rappresentazione dei tile su un immagine di esempio, evidenziando le sovrapposizioni tra i ritagli.

## 3.3 Prevenzione del Data Leakage nello Split dei Dati

Una delle sfide più insidiose nell'addestramento di modelli su dataset partizionati tramite *tiling* è il fenomeno del *Data Leakage*. Nella versione originale della pipeline, i ritagli generati venivano assegnati casualmente ai set di Addestramento (Train), Validazione (Valid) e Test. Poiché le *patch* adiacenti estratte da una singola immagine 4K presentano condizioni molto simili, questa suddivisione casuale può inquinare i set per testare.

Il modello, durante la fase di validazione, veniva testato su background e *feature* visive che aveva già parzialmente osservato in fase di addestramento, producendo metriche sovrastimate e non rappresentative della reale capacità di generalizzazione in produzione.

Il problema è stato risolto architetturalmente riscrivendo il flusso di preprocessing: lo *split* casuale (es. 80% Train, 10% Val, 10% Test) viene ora eseguito *a monte*, a livello delle immagini 4K originali. Solo successivamente le immagini assegnate ai rispettivi set vengono sottoposte al processo di tiling. Questo garantisce una suddivisione completa tra i set, rendendo le metriche finali oggettive e affidabili.

Nel Codice 3.1 è riportato un estratto dello script di preprocessing che mostra come lo split venga calcolato sull'immagine originale, garantendo l'assenza di Data Leakage tra i vari set. Lo script completo, comprendente le logiche di prevenzione della distorsione e calcolo dell'overlap, è disponibile integralmente nell'Appendice A.

```
    rand_val = random.random()
    if rand_val < PERCENTUALE_VAL:
        split_corrente = 'valid'
    elif rand_val < (PERCENTUALE_VAL + PERCENTUALE_TEST):
        split_corrente = 'test'
    else:
        split_corrente = 'train'
```

Listing 3.1: Assegnazione dello split a livello immagine

## 3.4 Bilanciamento del Dataset e Hard Negative Mining

Il processo di *tiling* su vaste aree di cantiere genera inevitabilmente un enorme sbilanciamento delle classi: la maggior parte dei ritagli inquadra cielo, asfalto o strutture vuote. In generale infatti, i dataset iniziali presentano più della metà di immagini considerate di puro *background* (senza annotazioni valide).

Fornire un dataset così sbilanciato alla rete causerebbe una convergenza del modello verso una classificazione *pigra*, in cui il modello massimizza la *Precision* a discapito della *Recall*, ignorando gli oggetti target. D'altra parte, eliminare totalmente il background renderebbe il modello suscettibile a un alto tasso di Falsi Positivi.

È stato quindi sviluppato uno script di *pruning* selettivo per scartare in modo casuale l'eccesso di immagini vuote, normalizzando il rapporto target/background a un valore ottimale del 10% (ridotto al 2% per il task specifico delle scritte pubblicitarie). Questa percentuale di immagini vuote rimanenti agisce come regolarizzatore, implementando la tecnica nota come *Hard Negative Mining*: il modello viene esplicitamente addestrato a osservare ad esempio elementi di disturbo tipici del cantiere (come secchi gialli o elementi circolari) senza classificarli erroneamente come elmetti (DPI).

## 3.5 Workflow di Annotazione e Strumenti di Labeling

Il processo di annotazione delle immagini (*labeling*) è una delle fasi più onerose e delicate. Per questo progetto è stata sfruttata una piattaforma cloud molto nota in ambito Computer Vision, *Roboflow*.

### 3.5.1 Roboflow e la gestione dei dataset Cloud

Nelle fasi iniziali, è stata utilizzata la piattaforma **Roboflow** (nella figura 3.3 un esempio d'uso), uno strumento *SaaS* (Software as a Service) che permette di gestire l'intero ciclo di vita del dato: dall'upload alla generazione di diverse versioni del dataset con trasformazioni

di *augmentation* applicate in tempo reale. Roboflow è stato fondamentale per attingere a dataset *open-source* già annotati, accelerando la fase di *cold start* dei modelli, ma anche nella fase di annotazione delle immagini provenienti dalle telecamere.

### 3.5.2 Workflow "Human-in-the-loop" con Roboflow

Per ottenere dei modelli *domain accurate* rispetto all'ambito di applicazione desiderato (cantieri), i dataset pubblici già annotati non sono sufficienti. Per questo motivo sono state scaricate in blocco immagini scattate dalle telecamere di TimeLapseLab in modo da ottenere un dataset formato da immagini specifiche per i cantieri.

Con l'aumentare della mole di dati proprietari da processare, però, si è resa necessaria una transizione verso un workflow più rapido. È stato adottato quindi un metodo chiamato *AI-assisted labeling* fornito direttamente da Roboflow, che, sfruttando i cosiddetti *Foundational Models* (modelli preaddestrati) ha generato delle *bounding box* preliminari. Queste sono state utilizzate per applicare un primo filtro alle immagini, e, considerato che in una cartella mensile si trovano migliaia di immagini, questo metodo ha permesso di ridurre drasticamente i tempi di annotazione, garantendo al contempo un'elevata qualità del dato finale.

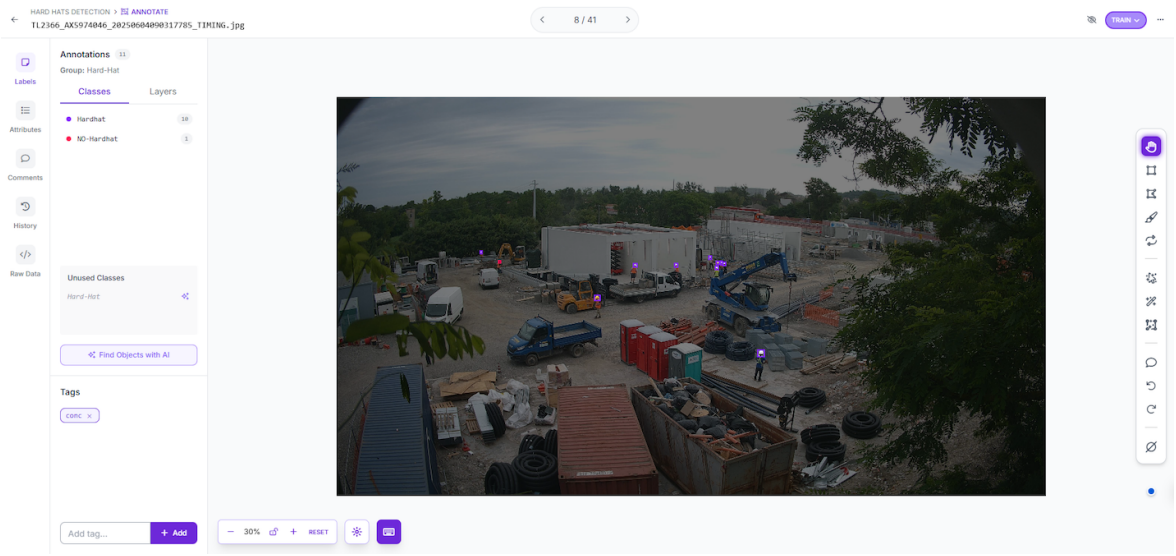



Figura 3.3: Esempio di utilizzo del software Roboflow per aggiungere annotazioni ad un immagine.

# Capitolo 4

# Addestramento e Validazione dei Modelli

In questo capitolo vengono descritte le fasi di sperimentazione, addestramento e validazione delle architetture di rete neurale. Il lavoro è stato suddiviso in task specifici, ciascuno caratterizzato da sfide di dominio uniche. Per ogni task, si illustrano il setup sperimentale, le strategie di *Data Engineering* adottate e l'analisi rigorosa delle metriche di valutazione al fine di giustificare le scelte architetturali finali per il deployment in produzione.

## 4.1 Task 1: Rilevamento dei Dispositivi di Protezione Individuale (DPI)

Figura 4.1: Esempio di immagine con caschetti oscurati.

Il primo obiettivo di questo lavoro è stato lo sviluppo di un modello in grado di rilevare l'utilizzo degli elmetti protettivi nei cantieri (un esempio nella figura 4.1), classificando i

soggetti in due categorie: personale equipaggiato (*Hardhat*) e personale non equipaggiato (*NO-Hardhat*).

### 4.1.1 Setup dell'Infrastruttura e Risoluzione Colli di Bottiglia

L'addestramento ha richiesto ingenti risorse computazionali. È stata utilizzata un'infrastruttura Cloud GPU basata su istanze vast.ai (dove è possibile noleggiare GPU a prezzi contenuti), equipaggiate con acceleratori grafici NVIDIA RTX 3090 (24GB di VRAM) ed eseguite all'interno di container Docker.

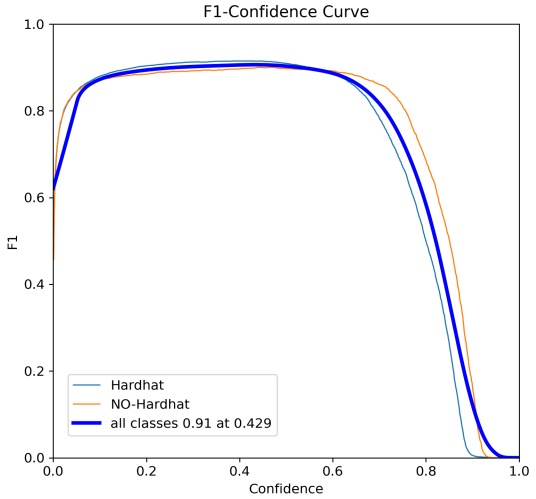
Durante le fasi iniziali di setup, è emerso un blocco silenzioso (*silent crash*) durante la fase di *data loading*. L'analisi del problema ha rivelato che il crash era causato dai limiti della memoria condivisa (`/dev/shm`) imposti di default dai container Docker. Il collo di bottiglia è stato risolto tramite un attento bilanciamento dei parametri di dataloader: si è proceduto a ridurre il *batch size* e ad azzerare il numero di *workers* dedicati al pre-caricamento parallelo, sacrificando parzialmente la velocità di iterazione per garantire la stabilità del processo su istanze non privilegiate.

```
    results = model.train(
    data="/workspace/dataset_caschetti/data.yaml",
    epochs=50,
    imgsz=640,
    batch=32,
    project=project_path,
    name='train_caschetti_v1',
    save=True,
    device=0,
    workers=8
)
```

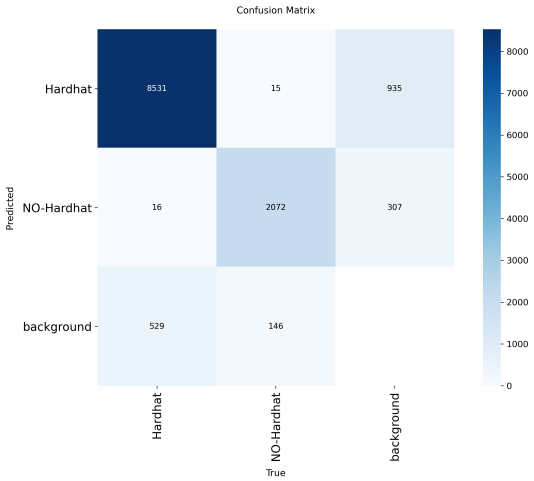
Listing 4.1: Parametri ottimizzati per il training su vast.ai

### 4.1.2 Primo Addestramento (Baseline) e Analisi degli Errori

Il primo ciclo di addestramento (Round 1) è stato eseguito utilizzando l'architettura **YOLOv11m** (Medium) per 50 epoche. Il modello ha raggiunto risultati preliminari eccellenti, registrando un mAP@0.5 del 93.6%, come visibile nella figura 4.3. L'analisi della curva F1-Confidence ha permesso di individuare una prima soglia di confidenza ottimale pari a 0.429, come da figura 4.2a.




(a) Curva F1 Score - Confidence




(b) Confusion Matrix

Tuttavia, un'analisi qualitativa della Matrice di Confusione (figura 4.2b) ha evidenziato un problema critico di *Domain Gap*. Il modello generava alcuni Falsi Positivi sistematici causati dal "rumore visivo" tipico dei cantieri: oggetti circolari o cromaticamente affini, come secchi gialli o elementi di tubature, venivano erroneamente classificati come elmetti.

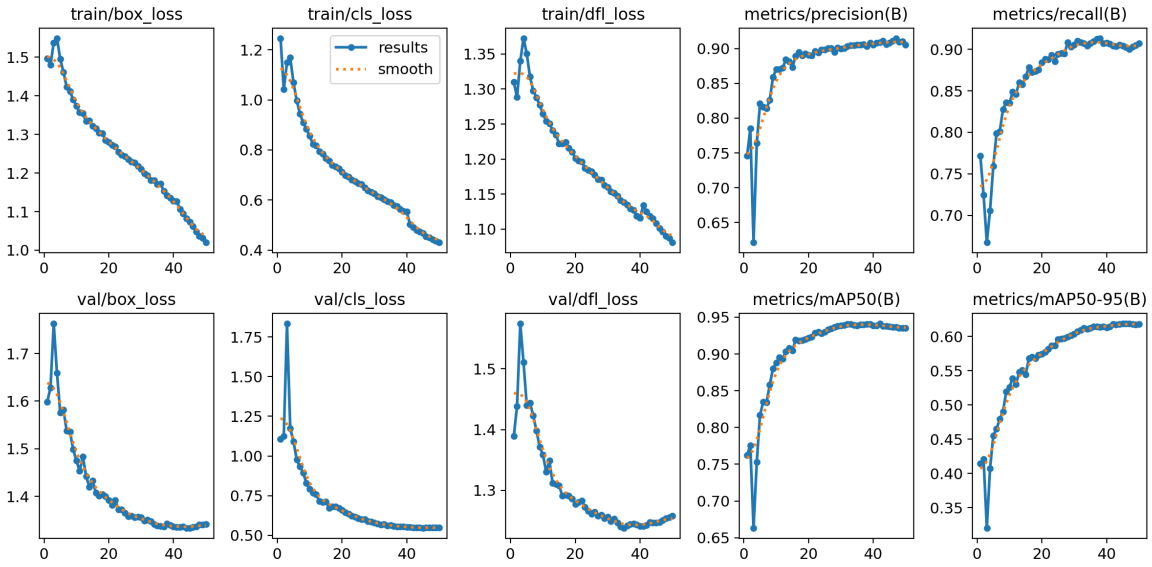



Figura 4.3: Analisi del training

### 4.1.3 Transizione verso un approccio Data-Centric

Per abbattere i Falsi Positivi, si è deciso di scartare l'ipotesi di modificare gli iperparametri della rete, adottando un approccio *Data-Centric*.

Per accelerare il processo di annotazione è stato sfruttato l'AI-labeling di Roboflow (come descritto nel capitolo 3). Questo ha abilitato una pipeline di *Human-in-the-loop Labeling*: l'Intelligenza Artificiale pre-annotava le nuove immagini proprietarie del cantiere, limitando l'intervento umano alla sola revisione e correzione degli errori, e filtrando immediatamente le immagini in cui non fossero presenti elementi rilevanti (in questo caso quindi caschetti).

È stata applicata una rigorosa strategia di *Hard Negative Mining*: si è deciso deliberatamente di conservare un 10-15% di immagini di puro *background* (contenenti gru, secchi gialli e attrezzature varie) per forzare la rete neurale a studiare le caratteristiche dei Falsi Positivi e imparare a discriminarli dai veri DPI. Il modello ha quindi subito un processo di *Fine-Tuning* su questo nuovo dataset ibrido, mantenendo tutta la varianza visiva del dataset originale.

### 4.1.4 Secondo Addestramento e Ottimizzazione del Modello

Il secondo addestramento, alimentato dal dataset ottimizzato geometricamente tramite *tiling* (come descritto nel capitolo 3) e arricchito con i *Negative Samples*, ha portato a un miglioramento delle performance, come da figura 4.4.

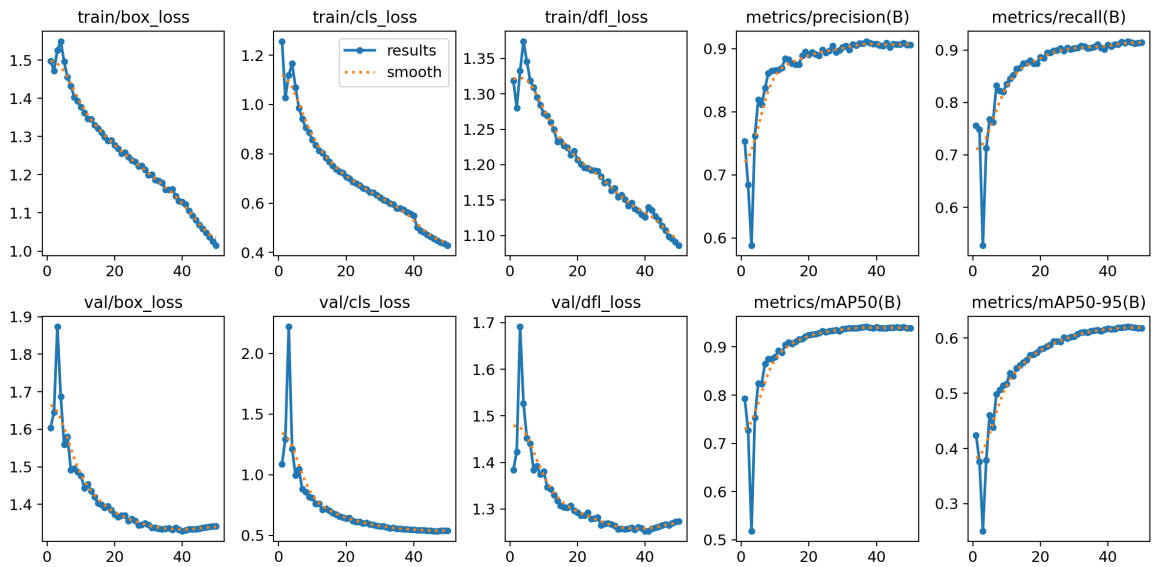
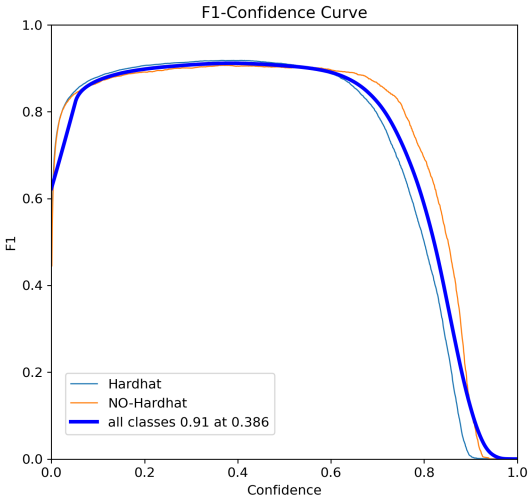



Figura 4.4: Analisi del training

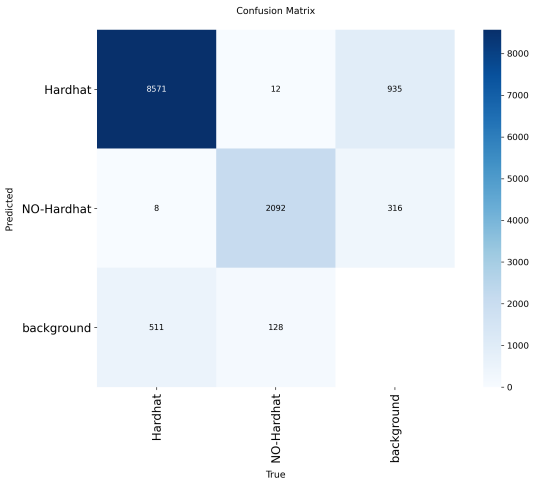
Le metriche finali hanno registrato un incremento del mAP@0.5, salito al **94.1%**. L'analisi comparativa della Matrice di Confusione (figura 4.5b) ha evidenziato:

- Una riduzione dei Falsi Negativi (operai a testa scoperta non rilevati).

- La confusione diretta tra le due classi target (casco scambiato per testa scoperta e viceversa) è scesa da 31 a sole 20 occorrenze su un campione di test di oltre 10.000 rilevamenti.




(a) Curva F1 Score - Confidence




(b) Confusion Matrix

## 4.2 Task 2: Rilevamento e Oscuramento delle Targhe Automobilistiche

Figura 4.6: Esempio di un immagine con targa offuscata. Da notare come la targa abbia dimensioni molto ridotte rispetto al resto dell'immagine.

Il secondo task affrontato riguarda l'identificazione e la successiva anonimizzazione delle targhe dei veicoli di passaggio ripresi dalle telecamere in zone di cantiere (un esempio

nella figura 4.6). Questo compito riveste un'importanza critica per la conformità al GDPR, richiedendo una *Recall* estremamente elevata per minimizzare il rischio di mancate rilevazioni.

### 4.2.1 Benchmarking e Analisi del Domain Gap

L'attività è iniziata con una fase di valutazione oggettiva (*benchmarking*) per testare l'efficacia di modelli pre-addestrati allo stato dell'arte, disponibili su piattaforme come HuggingFace, Kaggle e Roboflow. Per questa valutazione è stata sviluppata una pipeline di test custom basata sul calcolo delle metriche standard (TP, FP, FN) con una soglia IoU fissata a 0.5.

I risultati hanno evidenziato un marcato *Domain Gap*: i modelli generalisti hanno ottenuto una *Recall* insoddisfacente, compresa tra il 13% e il 26%. L'analisi qualitativa ha identificato le cause principali nelle peculiarità visive del dominio di TimeLapseLab:

- **Prospettiva Top-Down:** Le telecamere di cantiere, posizionate in alto, offrono angolazioni molto diverse dai dataset standard di *License Plate Detection* generici.
- **Condizioni Ambientali:** Presenza di ombre nette, riflessi metallici e occlusioni parziali dovute a recinzioni da cantiere.

### 4.2.2 Data Engineering e Strategie di Campionamento

Per colmare il divario prestazionale, si è proceduto alla creazione di un dataset custom. Un aspetto critico è stato evitare il *Data Leakage* e l'*overfitting* sui veicoli statici (ad esempio, mezzi parcheggiati per giorni nella stessa posizione).

È stata adottata una strategia di *campionamento intelligente*: invece di estrarre frame sequenziali, sono stati selezionati frame distanziati nel tempo per massimizzare la varianza visiva (cambiamenti di luce, condizioni atmosferiche e ombre proiettate), istruendo il modello a riconoscere la struttura della targa anziché le caratteristiche del background statico.

Le immagini 4K sono state processate tramite *tiling* (griglia 3X3, come analizzato nella figura 3.2) per generare *patch* compatibili con l'input di YOLO senza perdere risoluzione. Il dataset è stato poi bilanciato tramite uno script di *pruning* che ha ridotto la percentuale di immagini vuote al 10%, prevenendo cali di sensibilità della rete.

### 4.2.3 Risoluzione di Criticità Tecniche: Il Problema dello Spazio Colore

Durante l'integrazione del modello con la libreria SAHI, è emersa una criticità tecnica che azzerava le rilevazioni nonostante l'addestramento fosse andato a buon fine. L'analisi del problema ha rivelato un disallineamento nello spazio colore: la libreria OpenCV carica nativamente le immagini in formato **BGR**, mentre i modelli YOLO e la pipeline di inferenza richiedono il formato **RGB**. Questa inversione di canali cromatici alterava completamente la

percezione delle *feature* da parte della rete neurale. Il problema è stato risolto implementando una conversione esplicita (`cv2.cvtColor`) prima della fase di inferenza, ripristinando le performance attese.

### 4.2.4 Training

La fase finale di training ha previsto un approccio basato sull'utilizzo di diversi dataset, uniti alle immagini nei cantieri. Sono stati testati due diversi dataset, ottenuti da Roboflow, differenti nelle tipologie di targhe (europee o americane), nel contesto e nel numero di immagini. Per valutare i dataset sono stati inizialmente addestrati due modelli (sempre YOLOv11m) su di essi, ed è stato addestrato anche un modello solamente sulle immagini estratte dai cantieri.

Per ottenere il modello definitivo è stato creato un dataset unico sfruttando il dataset con targhe europee unito a quello specifico dei cantieri. Il secondo dataset è stato scartato in quanto, oltre ad ottenere anche targhe americane (non utili nel contesto di TimeLapseLab), il numero di immagini nel dataset superava le 10000 unità, numero troppo elevato rispetto alle 1000 circa dai cantieri.

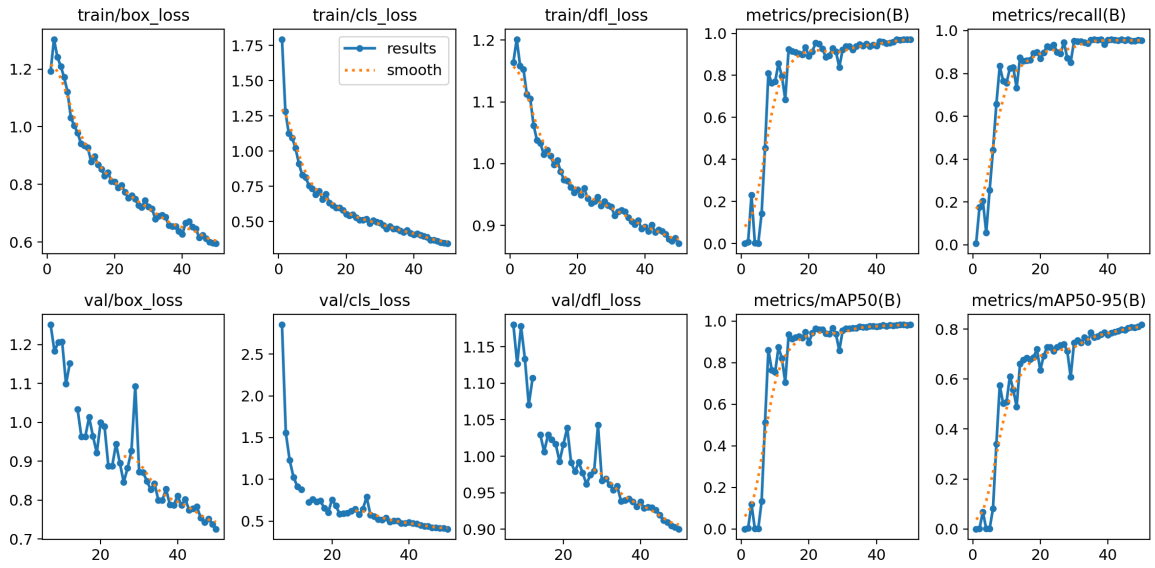
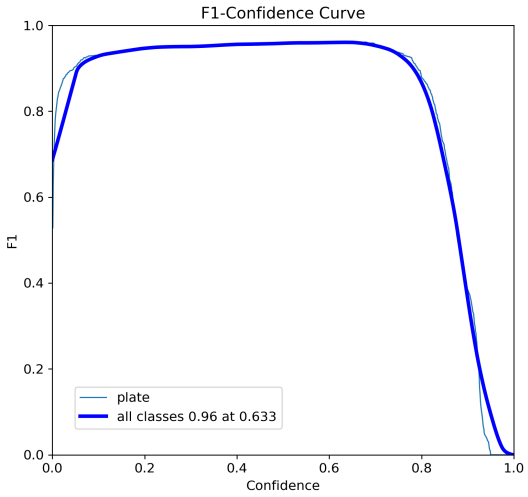
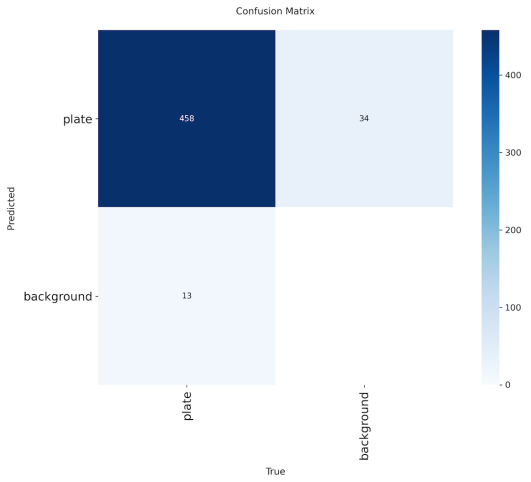



Figura 4.7: Analisi del training

Il modello finale ha raggiunto ottimi risultati, con un F1 score di 0.96 alla soglia di 0.63 di confidenza (visibile nella figura 4.8b), un mAP@0.5 al **98.1%** e un mAP@0.5-0.95 al **81.7%** (come da grafici nella figura 4.7). Inoltre ha mostrato ottimi risultati anche nel caso della matrice di confusione, con appena 13 targhe scambiate per background e 34 falsi positivi (come da figura 4.8b).




(a) Curva F1 Score - Confidence




(b) Confusion Matrix

## 4.3 Task 3: Rilevamento e Anonimizzazione di Persone

Figura 4.9: Esempio di immagine con persone oscurate. Da notare che viene rilevata anche la persona piegata in basso.

Il terzo task ha riguardato lo sviluppo di una pipeline per l'individuazione di sagome umane all'interno di un dataset composto da oltre 6.600 immagini 4K (un esempio nella figura 4.9). L'obiettivo primario era garantire la privacy dei soggetti ripresi prima dell'archiviazione dei *frame*, rendendo necessario un rilevamento robusto anche in condizioni di forte distanziamento dalla telecamera.

### 4.3.1 Evoluzione Architetturale: Da Cloud API a Local Deployment

In una prima fase esplorativa, l'inferenza è stata gestita tramite chiamate API remote utilizzando l'*Inference-SDK* di Roboflow, appoggiandosi a modelli hostati in Cloud. Tuttavia, i limiti imposti dalle licenze gratuite e gli errori di rete frequenti (`HTTP 403 Forbidden`) per chiamate al di fuori del *workspace* hanno reso l'architettura instabile e inadatta all'elaborazione massiva di oltre 6.000 immagini.

Per garantire indipendenza e abbattere la latenza di rete, l'architettura è stata completamente riscritta per operare in locale. Si è proceduto all'installazione di librerie a basso livello (`pycuda` e `onnxruntime-gpu`) all'interno dell'ambiente di elaborazione (Google Colab con GPU T4/L4). I pesi del modello (`.pt`) sono stati scaricati e istanziati direttamente sulla GPU locale, permettendo al sistema di azzerare i costi API e massimizzare il *throughput* per *frame*. In questa fase è stato inoltre necessario aggiornare la pipeline di *parsing* per supportare Pydantic V2, passando dai metodi deprecati `.dict()` ai moderni `.model_dump()` per l'estrazione dinamica delle classi.

### 4.3.2 Integrazione di SAHI e Debugging della Memoria GPU

Così come nei task precedenti, l'utilizzo nativo di immagini 4K portava a una drastica perdita di dettaglio post-resize. L'integrazione di SAHI è stata adottata con *patch* da 1920x1080 pixel e un *overlap* del 15%, garantendo un compromesso ottimale tra velocità (elaborando 6-8 *tile* per foto in circa 1.33 secondi totali) e precisione.

Tuttavia, durante la fase di *slicing*, è emerso un blocco architetturale severo: la GPU non riusciva a elaborare i ritagli generati da SAHI, restituendo in output tensori vuoti (zero rilevamenti). L'analisi a basso livello ha rivelato che la causa risiedeva nella gestione della memoria da parte della libreria `NumPy`. I ritagli generati erano viste (*views*) in memoria di porzioni più ampie dell'immagine e risultavano, pertanto, allocati in aree di memoria *non contigue*. Il motore ONNX/YOLO, che richiede vettori contigui per il trasferimento parallelo alla memoria VRAM della GPU, falliva dietro le quinte.

L'errore è stato risolto inserendo una trasformazione esplicita sulla singola *patch* prima di inoltrare il tensore al modello neurale, ristabilendo il corretto flusso di elaborazione, tramite la funzione `np.ascontiguousarray()`.

### 4.3.3 Scelta del Modello: Pre-addestrati e Fine-Tuned

Data l'enorme varianza intrinseca della classe *Persona* e l'estesa disponibilità di dataset pubblici sul tema, i primi test effettuati su modelli addestrati custom per il cantiere si sono rivelati insufficienti nella generalizzazione rispetto ai modelli allo stato dell'arte.

Si è optato quindi per la creazione di un dataset combinato seguendo lo stesso procedimento dei task precedenti, ovvero è stato selezionato un dataset pubblico estratto da Roboflow ed unito con immagini etichettate estratte dalle telecamere nei cantieri.

### 4.3.4 Risultati del Training

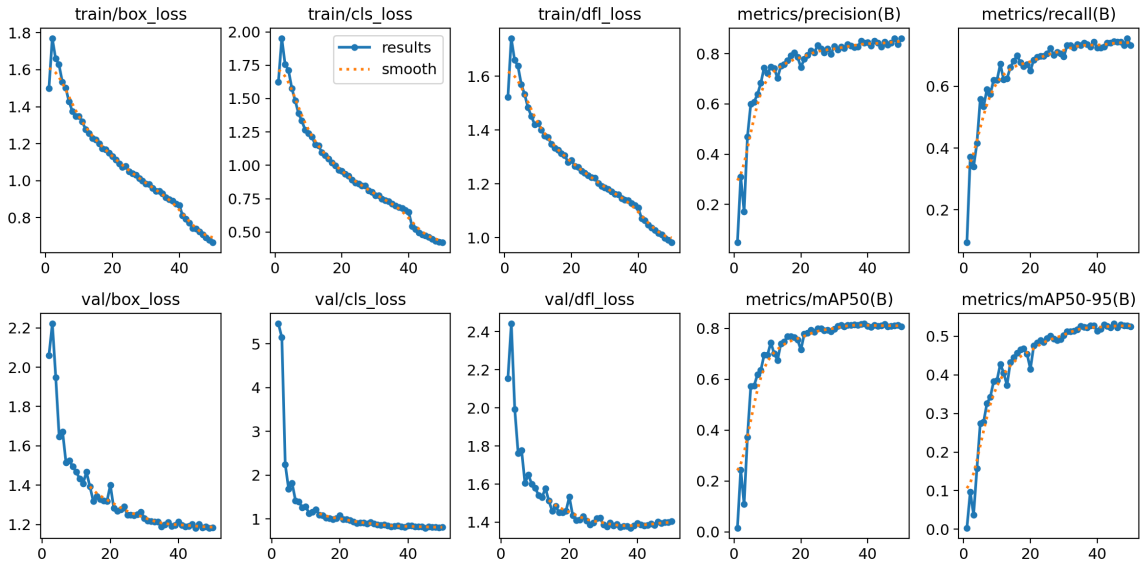
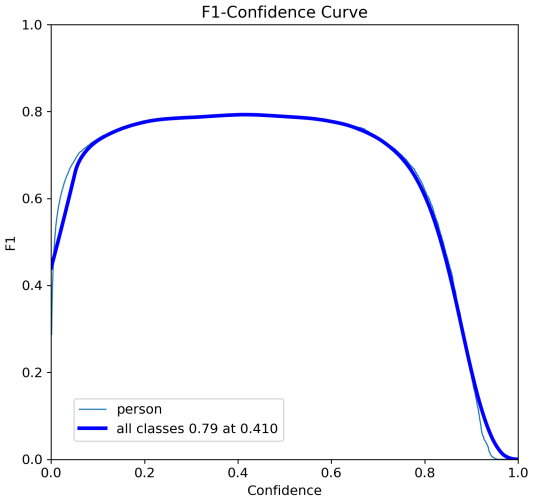
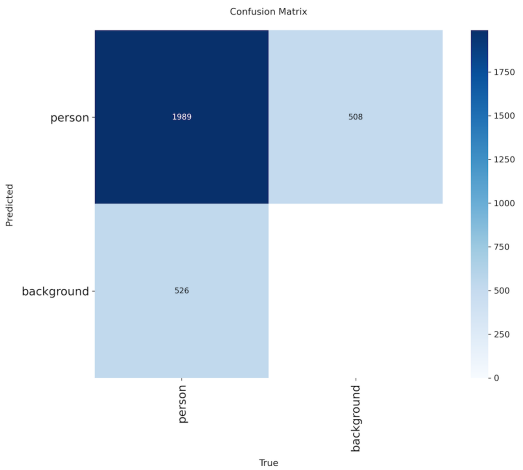



Figura 4.10: Analisi del training

Il modello ottenuto ha raggiunto buoni risultati, toccando l'81% di mAP@0.5 (come da figura 4.10), un valore di F1 score pari a 0.8 (in figura 4.11a) e una confusion matrix che riporta un basso numero di falsi positivi e falsi negativi (come visibile nella figura 4.11b).




(a) Curva F1 Score - Confidence




(b) Confusion Matrix

## 4.4 Task 4: Rilevamento di Grafiche Pubblicitarie e Loghi Aziendali


Figura 4.12: Esempio di immagine con grafiche pubblicitarie oscurate.

L'ultimo task affrontato riguarda l'identificazione di scritte pubblicitarie e loghi presenti sulle fiancate dei veicoli commerciali (un esempio nella figura 4.12). Questo obiettivo, sebbene apparentemente simile alla rilevazione di targhe, ha presentato sfide uniche legate alla notevole variabilità del dominio. Infatti, a differenza per esempio di targhe o caschetti, le grafiche pubblicitarie non riportano una costanza in dimensioni e colore. Per questo motivo questo task è risultato molto più complesso, partendo dalla selezione del dataset preesistente fino all'ottenimento di buoni risultati.

### 4.4.1 Ridefinizione del Problema: dalla Detection alla Redaction

Inizialmente, l'obiettivo era stato ipotizzato come un compito di *Optical Character Recognition* (OCR) per l'estrazione del testo. Tuttavia, un'analisi più approfondita dei requisiti di privacy ha portato a un cambio di paradigma: l'obiettivo finale non è la lettura del contenuto, ma la sua *Redaction* (oscuramento).

Di conseguenza, il task è stato semplificato in una classificazione a classe singola (`adv`), dove il *bounding box* non deve circondare le singole parole, ma l'intera area grafica o il logo. Questo approccio ha permesso di semplificare l'architettura del modello, focalizzando l'addestramento sulla capacità di distinguere superfici testuali complesse dal background rappresentato dalla carrozzeria del veicolo.

### 4.4.2 Sperimentazione con Dataset Esterni e Rumore di Dominio

Sono stati condotti esperimenti iniziali integrando dataset *open-source* come **COCO-Text** e dataset specifici per la *Logo Detection*. Tuttavia, l'analisi delle metriche ha evidenziato risultati insoddisfacenti, con un *F1-Score* bloccato attorno al 42-50%.

Il fallimento è stato attribuito alla notevole eterogenità del dominio: i dataset pubblici sono spesso composti da immagini *sceniche* (cartelli stradali, vetrine), mentre il dominio di TimeLapseLab è caratterizzato da telecamere fisse con prospettive dall'alto. Inoltre, l'addestramento su singoli caratteri portava la rete a generare un numero elevato di Falsi Positivi su texture ripetitive del cantiere (es. griglie o reti). Si è quindi deciso di puntare esclusivamente su un dataset custom proprietario.

### 4.4.3 Ottimizzazione del Preprocessing e Data Leakage

Durante la creazione del dataset custom, sono stati risolti due problemi critici nel codice di preparazione:

1. **Risoluzione del Data Leakage:** Come discusso nel Capitolo 3, la versione originale dello script divideva i *tile* in modo casuale, inquinando i set di validazione con porzioni della stessa immagine presente nel training set. La riscrittura dello script ha garantito la purezza dei dati.

2. **Pruning Estremo del Background:** Data la rarità dei furgoni pubblicitari rispetto all'area totale del cantiere, lo script di *tiling* generava un eccesso di fette prive di informazioni. La probabilità di conservare *tile* di background è stata abbassata al **2%**, forzando il modello a specializzarsi sulle *feature* dei veicoli commerciali.

### 4.4.4 Inference Innovation: TTA e Monkey Patching

Per massimizzare la capacità di rilevamento su veicoli lontani o angolati, è stata implementata la **Test-Time Augmentation (TTA)**. Tuttavia, è emerso un limite tecnico: la libreria SAHI non supportava nativamente il passaggio del parametro `augment=True` al motore di inferenza di YOLOv11.

Per superare questo ostacolo senza attendere aggiornamenti ufficiali della libreria, è stata utilizzata la tecnica del **Monkey Patching**. È stato sviluppato un *Custom Wrapper* (come nel codice 4.2) in Python capace di intercettare a runtime le chiamate interne di SAHI e iniettare forzatamente il parametro di *augmentation* in ogni singola *patch* processata. Questo intervento ha permesso di innalzare l'accuratezza del modello senza modificare il codice sorgente della libreria esterna.

```
1 class YoloTTAWrapper:
2     def __init__(self, model):
```

```
        self.model = model

    def __call__(self, *args, **kwargs):
        kwargs['augment'] = True  # Test-Time Augmentation Forzata
        return self.model(*args, **kwargs)

    def __getattr__(self, item):
        return getattr(self.model, item)
```

Listing 4.2: Wrapper Custom per abilitazione TTA

### 4.4.5 Analisi dei Risultati Finali

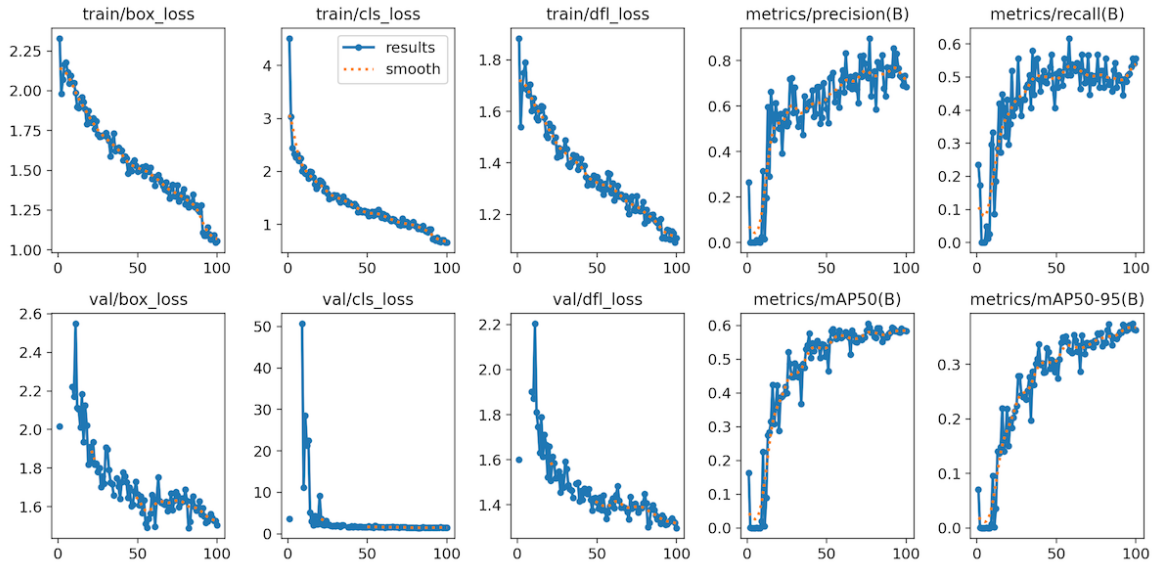
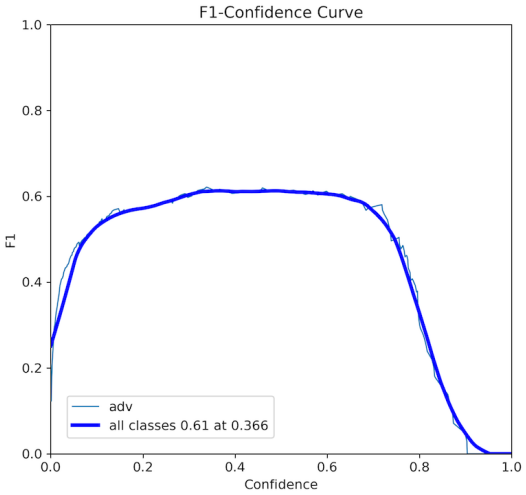
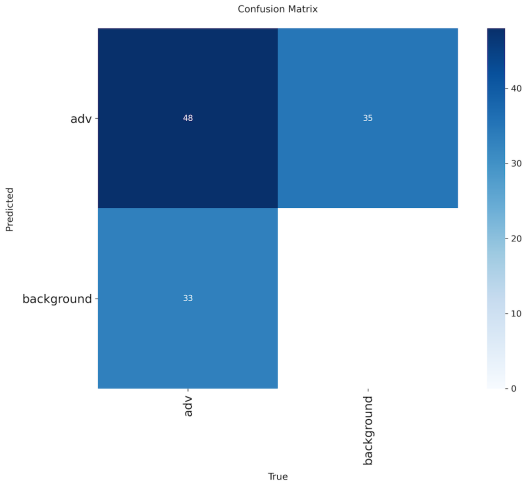



Figura 4.13: Analisi del training

Il passaggio al dataset unicamente composto da immagini nel contesto dei cantieri, unito a una risoluzione di training maggiorata (`imgsz=800`) e all'uso combinato di TTA e SAHI, ha prodotto un incremento prestazionale netto. L'F1-Score è passato dal 50% iniziale al **61%** (come in figura 4.14a), dimostrando che la specializzazione sul dominio aziendale e le ottimizzazioni in fase di inferenza hanno permesso di creare un modello migliore. D'altra parte è evidente come la complessità del dominio, come visibile nella figura 4.13 e 4.14b, abbia impattato nella mAP e nella dimensione del dataset, e di conseguenza nella difficoltà di generalizzazione da parte del modello ottenuto.




(a) Curva F1 Score - Confidence




(b) Confusion Matrix

# Capitolo 5

# Progettazione del Backend e dell'Inference Server

In questo capitolo viene dettagliata la progettazione, lo sviluppo e l'ottimizzazione dell'infrastruttura software responsabile della messa in produzione dei modelli di Deep Learning. L'obiettivo principale è stato creare un *Inference Server* robusto, capace di elaborare le immagini provenienti dalle telecamere aziendali attraverso le reti YOLO sviluppate per l'Object Detection e i successivi algoritmi di anonimizzazione (*Obfuscation*), senza introdurre colli di bottiglia nel flusso dati principale.

## 5.1 Architettura di Sistema: L'Approccio Ibrido

Il sistema originario prevedeva una comunicazione diretta e sincrona tra l'Executor (il modulo responsabile della gestione del flusso logico delle immagini) e il Web Server incaricato dell'inferenza. Tuttavia, l'elaborazione di reti neurali su immagini 4K tramite *tiling* è un'operazione *compute-intensive*. Mantenere una connessione HTTP sincrona aperta per l'intera durata dell'inferenza esponeva il sistema a gravi rischi di *timeout* e di esaurimento dei *thread* disponibili sul server.

Per risolvere questa criticità, il sistema è stato riprogettato disaccoppiando la ricezione delle richieste dall'elaborazione vera e propria, adottando un'architettura ibrida HTTP/MQTT.

Come illustrato nella Figura 5.1, il nuovo flusso si articola come segue:

1. **Web Server Asincrono (HTTP):** Sviluppato in Python con la libreria `aiohttp`, riceve le richieste di elaborazione tramite chiamate `POST` dall'Executor. Si occupa esclusivamente di generare un *payload* JSON contenente i metadati (es. percorso del file, parametri di *blur* e soglie di confidenza).

2. **Gestione Asincrona (MQTT):** Invece di eseguire il modello, il Web Server pubblica il *job* su un *topic* MQTT dedicato (tramite broker Mosquitto) e si pone in attesa

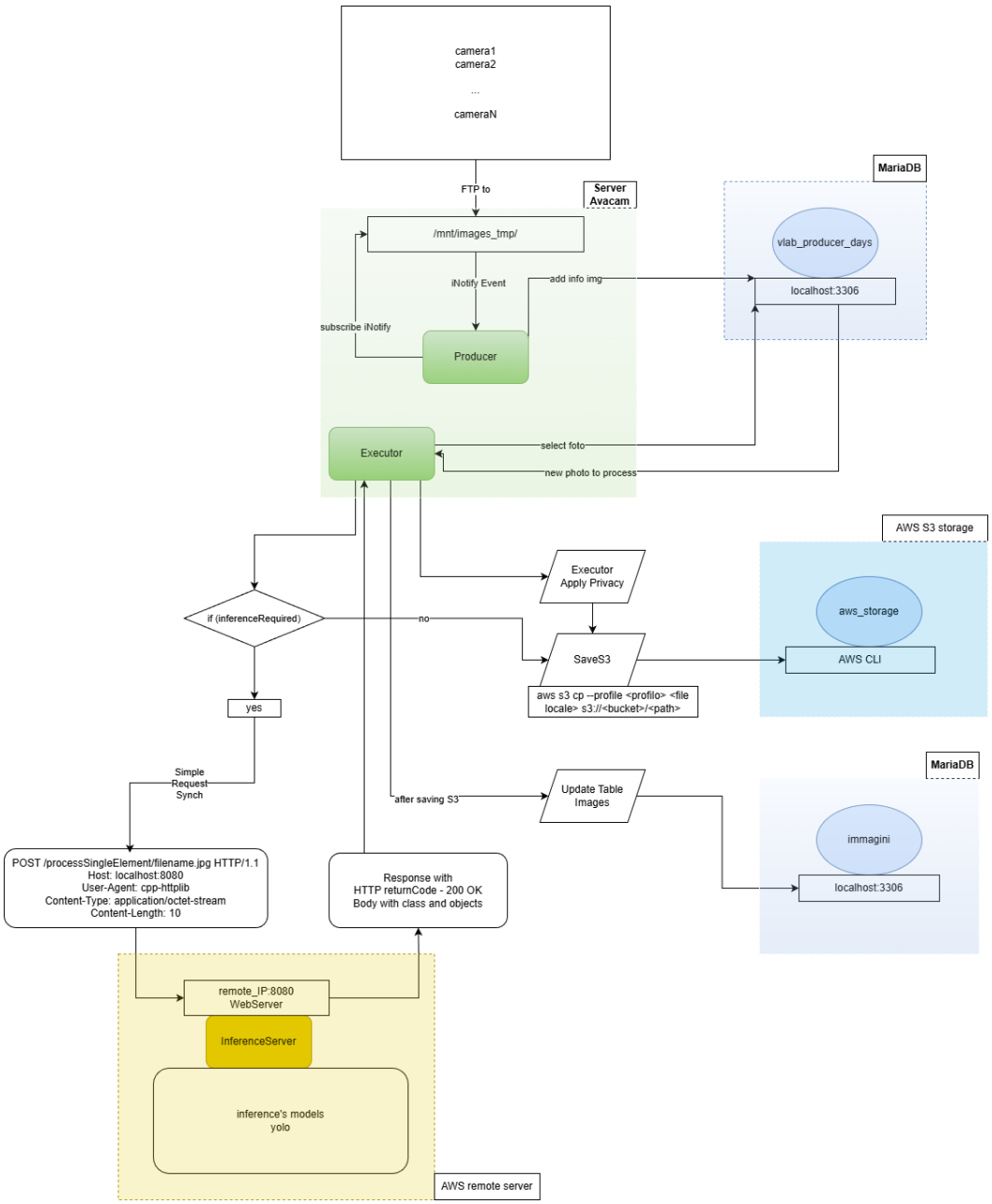



Figura 5.1: Architettura di sistema di TimeLapseLab. Il riquadro giallo (*InferenceServer*) evidenzia il modulo ottimizzato durante il tirocinio per la gestione asincrona delle inferenze.

passiva (`await asyncio.wait_for`) su un *topic* di risposta legato all'ID univoco della richiesta.

3. **Worker Indipendente:** Un processo in *background* isolato (`inference_worker.py`) ascolta le notifiche MQTT, esegue l'inferenza YOLO tramite SAHI, applica le maschere di offuscamento e, a lavoro concluso, pubblica la risposta sbloccando la chiamata HTTP originaria.

Nel Codice 5.1 è riportato un frammento del gestore HTTP (`ws_handler_2.py`) che illustra il disaccoppiamento: il server pubblica il *payload* su MQTT e sospende l'esecuzione della richiesta corrente ponendosi in ascolto asincrono tramite la funzione `asyncio.wait_for`, liberando così il *thread* principale.

```
    # 3. INVIO ORDINE AL WORKER
    logger.info(f"Delegando lavoro al worker per ID: {req_id}")
    # Publish su "Notifiche" di tutto il pacchetto dati
    client.publish("Notifiche", json.dumps(job_payload))

    # 4. ATTESA PASSIVA
    logger.info(f"In attesa del Worker...")
    try:
        # Richiesta bloccata finche' future non e' 'set_result'
        # Aggiunta del timeout di 20 secondi per prevenire lock infiniti
        result_json = await asyncio.wait_for(fut, timeout=20.0)
        return web.Response(status=200, text=f"Response about {req_id} -
    inferenza eseguita")

    except asyncio.TimeoutError:
        logger.warning(f"Timeout inferenza scaduto per {req_id}")
        return web.Response(status=504, text=f"Inferenza fallita")
```

Listing 5.1: Gestione asincrona dell'attesa tramite Future e MQTT

Per comprendere l'alto livello di flessibilità e configurabilità della pipeline sviluppata, il Codice 5.2 mostra un esempio del payload JSON inviato tramite richiesta HTTP POST all'Inference Server. La struttura del messaggio riflette la filosofia del sistema, concepito per accoppiare la trasmissione dei dati visivi alla definizione dinamica dei parametri algoritmici di esecuzione.

```
{
    "id" : "image_name",
    "inferencePars" : "person=0.5, plate=0.6",
    "blurConfigs": {
        "plate": {
            "forma_maschera": "rettangolare",
            "tipo_maschera": "blur",
```

```
            "kernel_size": 61,
            "paint_grayscale_value": 125,
            "area_multiplier": 1.2
        },
        "person": {
            "forma_maschera": "ellittica",
            "tipo_maschera": "paint",
            "kernel_size": 31,
            "paint_grayscale_value": 200,
            "area_multiplier": 2.0
        }
    }
}
```

Listing 5.2: Esempio di payload JSON utilizzato dall'inference server

L'analisi dettagliata dei campi evidenzia le seguenti funzionalità:

- `id`: Svolge una doppia funzione, infatti rappresenta innanzitutto una stringa che identifica univocamente il frame all'interno del sistema e che verrà utilizzata come chiave di tracciamento per i messaggi scambiati sul broker MQTT. Inoltre essa è anche il nome univoco (es *TLBVF2406_FK0399962_20250728000237516_TIMING.jpg*) dell'immagine ad alta risoluzione salvata sul file system condiviso. Come discusso in precedenza, si è optato per questo approccio per alleggerire i successivi passaggi sul broker MQTT.

- `inferencePars`: Consente la modifica dinamica e a runtime delle soglie di confidenza dei singoli modelli di rilevamento (ad esempio, impostando la confidenza per la classe *person* a 0.5 e per *plate* a 0.6), evitando la necessità di ricompilare o riavviare i moduli di inferenza. Inoltre, specificando le classi da rilevare, vengono inizializzati e utilizzati solo i modelli relativi a quelle specifiche classi (in questo caso *plate detection* e *person detection*), ottimizzando quindi le tempistiche di inferenza.

- `blurConfigs`: Oggetto nidificato che definisce le specifiche geometriche e matematiche dei filtri di offuscamento da applicare sulle *bounding box* predette, segmentate per classe di appartenenza:

  - `forma_maschera`: Specifica la geometria della maschera di oscuramento (*rettangolare* per le targhe, adattandosi alla morfologia standard del target, ed *ellittica* per le sagome umane).
  - `tipo_maschera`: Definisce l'operazione sui pixel; il valore *blur* innesca l'applicazione di un filtro di convoluzione gaussiana, mentre il valore *paint* esegue

una sovrascrittura a tinta unita basata sul valore di grigio definito nel successivo parametro, `paint_grayscale_value`.

- `kernel_size`: Rappresenta la dimensione del filtro di convoluzione (deve essere un numero dispari), che determina l'intensità della sfocatura.
- `area_multiplier`: Un coefficiente moltiplicativo che espande geometricamente i confini della maschera oltre il perimetro teorico della *bounding box* predetta (es. un incremento del 20% per le targhe e del 100% per le persone), garantendo la copertura dei dettagli biometrici periferici in piena conformità ai requisiti di *Privacy-by-Design*.

## 5.2 Configurazione del Tiling con SAHI in Produzione

Per individuare oggetti di piccole dimensioni (come targhe o DPI in lontananza) sulle immagini ad alta risoluzione in produzione, è stata integrata la libreria SAHI (*Slicing Aided Hyper Inference*). È stato fondamentale allineare i parametri di inferenza con quelli utilizzati matematicamente durante la fase di training (descritta nel Capitolo 3).

I parametri sono stati configurati nel modo seguente:

- **Dimensioni Tile (Slice):** 1400x800 pixel.
- **Overlap Ratio:** Calcolato matematicamente per garantire una sovrapposizione del 15% sull'asse Y (120 pixel) e di 180 pixel sull'asse X, prevenendo un possibile taglio delle *bounding box* poste sui bordi della finestra di scorrimento.
- Il *resize* dei singoli *tile* è applicato internamente dal modello YOLO, che scala le porzioni 1400x800 alla risoluzione nativa del tensore di addestramento (640x640 o 800x800 a seconda del modello).

## 5.3 Debugging e Risoluzione di Bug Architetturali

Durante la messa a punto dell'Inference Server, sono emerse diverse problematiche architetturali, risolte tramite interventi di *debugging* mirato.

### 5.3.1 Ottimizzazione del Payload MQTT (Size Limit)

Inizialmente, il sistema tentava di passare l'intera immagine codificata in stringa *Base64* all'interno del *payload* MQTT. Questa scelta causava *timeout* frequenti e fallimenti silenziosi dei messaggi, in quanto il peso superava i limiti ottimali del *broker*. La logica è stata riprogettata: il *Worker* riceve ora solo i metadati e il percorso del file. È il *Worker* stesso a caricare

l'immagine dal disco locale condiviso tramite `cv2.imread`, abbattendo drasticamente la latenza di rete e il consumo di RAM.

La transizione dall'elaborazione *in-memory* tramite Base64 alla lettura diretta da *file system* è visibile nel Codice 5.3, estratto dal *Worker* di inferenza (`inference_worker.py`).

```
    # 1. CARICAMENTO IMMAGINE
    # [DEPRECATO] Decodifica base64 -> Bytes -> NumPy Array -> OpenCV
    """
    img_bytes = base64.b64decode(base64_image)
    np_arr = np.frombuffer(img_bytes, np.uint8)
    img = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
    """

    # [NUOVA LOGICA] Lettura diretta da percorso condiviso per
    alleggerire MQTT
    image_path = OUTPUT_DIR / req_id
    img_bgr = cv2.imread(str(image_path))

    if img_bgr is None:
        logger.error(f"Impossibile leggere l'immagine per l'ID: {req_id}"
    )
        mqtt_client.publish(req_id, json.dumps({"status": "ERROR"}))
        return

    # Conversione Spazio Colore per compatibilita YOLO
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
```

Listing 5.3: Sostituzione della decodifica Base64 con lettura diretta da disco

### 5.3.2 Gestione Dinamica della Confidenza e Cross-Platform

Sono stati corretti bug di compatibilità tra ambienti Linux (server) e Windows (sviluppo), in particolare legati alla gestione dei percorsi generati dalla libreria `pathlib` (risolti tramite un *casting* esplicito a stringa prima delle chiamate OpenCV) e all'interruzione pulita del *loop* asincrono, prevenendo l'eccezione `NotImplementedError` su Windows. Inoltre, è stata ripristinata la logica di *parsing* per la gestione dinamica della confidenza. Il sistema originario sovrascriveva in modo errato le soglie di default di SAHI con valori imposti dal *client* eccessivamente stringenti (es. 0.6), causando problemi di rilevazione.

### 5.3.3 Allineamento degli Spazi Colore (RGB vs BGR)

Un bug critico e insidioso causava il totale fallimento delle rilevazioni in produzione rispetto ai test su Google Colab. L'analisi ha svelato un disallineamento dello spazio cromatico: OpenCV legge le immagini in formato **BGR**, mentre le reti YOLO, addestrate sui *tensor* Py-

Torch, attendono un input in formato **RGB**. L'inversione dello spettro generava *feature maps* illeggibili per la rete. L'inserimento della conversione esplicita `cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)` ha ripristinato l'accuratezza del 100%.

## 5.4 Analisi delle Prestazioni e Scalabilità

Per validare la robustezza del sistema, è stato sviluppato un *client* simulato (`Client.py`) in grado di generare carichi di lavoro concorrenti. Dopo aver rimosso *delay* artificiali ereditati da versioni precedenti (`asyncio.sleep`), è stato condotto un *benchmarking* preliminare.

I test in ambiente locale (sprovvisto di accelerazione hardware CUDA e basato su pura esecuzione CPU) hanno registrato un tempo di elaborazione medio di 6-7 secondi per singola immagine 4K. Tale tempistica, considerata l'elevata densità di inferenze generate dal *tiling* di SAHI (circa 6-9 passaggi per frame), dimostra un'eccellente ottimizzazione del codice Python. Un primo *deployment* in produzione sul server aziendale dotato di GPU dedicata ha permesso di testare le tempistiche reali, che si sono assestate sul secondo e mezzo per immagine, confermando quindi l'efficienza della pipeline sviluppata, pronta per essere utilizzata in un contesto industriale.

# Capitolo 6

# Risultati Sperimentali e Conclusioni

Il presente capitolo chiude l'elaborato illustrando i risultati finali ottenuti dall'integrazione dei modelli di rilevamento di oggetti all'interno dell'architettura di produzione. Vengono inoltre discusse le prestazioni computazionali del sistema, tratte le conclusioni sul lavoro svolto e delineati i possibili sviluppi futuri.

## 6.1 Validazione del Sistema Integrato e Risultati Qualitativi

L'obiettivo primario del progetto era la realizzazione di una pipeline end-to-end capace di garantire la *Privacy-by-Design* nei flussi video di TimeLapseLab. La validazione del sistema integrato ha confermato il raggiungimento di questo traguardo.

Il modulo di inferenza, istruito a partire dai modelli **YOLOv11** addestrati in modo custom per il dominio dei cantieri, ha dimostrato un'elevata robustezza. L'integrazione del motore **SAHI** ha permesso di mantenere intatta la capacità di rilevamento sulle immagini 4K, mitigando il problema della perdita di *feature* per oggetti distanti.

Dal punto di vista qualitativo, l'algoritmo di *obfuscation* (basato sull'espansione dell'area della *bounding box* tramite un moltiplicatore di tolleranza, ad esempio 1.2 per le targhe) si è rivelato fondamentale per garantire la compliance al GDPR.

Come visibile nella Figura 6.1, il filtro di sfocatura gaussiana (*Gaussian Blur*) copre non solo l'oggetto target, ma anche i dettagli biometrici periferici o le variazioni prospettiche dei loghi, rendendo l'anonimizzazione irreversibile prima del salvataggio nel database centrale aziendale.

Figura 6.1: Esempio di output del sistema: nel caso specifico è stata utilizzata una maschera ellittica con moltiplicatore 1.2 e valore di grigio pari a 61.

## 6.2 Analisi Prestazionale e Tempi di Latenza

In ambito *MLOps*, l'accuratezza di un modello deve essere bilanciata con i costi computazionali e i tempi di latenza del server. Il sistema ibrido HTTP/MQTT ha brillantemente superato gli *stress test* locali e su server aziendali.

Utilizzando un client di simulazione asincrono, è stato calcolato il tempo di latenza *End-to-End* per singola immagine 4K. In un ambiente privo di accelerazione hardware (esecuzione pura su CPU), il tempo medio di elaborazione (che include ricezione dati, caricamento da disco, *tiling* SAHI con 6-9 iterazioni, inferenza YOLO, offuscamento e invio della risposta MQTT) si attesta tra i **6 e i 7 secondi**, mentre grazie a test preliminari sul server aziendale, le tempistiche si sono ridotte a **1.5 secondi** per immagine.

Questo risultato certifica l'estrema ottimizzazione del codice Python e della gestione della memoria. Il disaccoppiamento architetturale garantisce inoltre che il *Web Server* principale non esaurisca i propri *thread* durante questi secondi di calcolo, delegando il carico in modo asincrono.

## 6.3 Conclusioni

L'attività svolta ha permesso di risolvere una problematica industriale complessa, passando da una prototipazione in Cloud (fortemente limitata da API esterne e *Data Leakage*) a un sistema distribuito *on-premise* proprietario, scalabile e rigorosamente valutato.

I contributi principali di questa tesi si possono riassumere in:

1. **Data Engineering:** Creazione di pipeline di *tiling* avanzate e risoluzione di colli di bottiglia legati all'allocazione di memoria (memoria condivisa Docker e *array* contigui in GPU).

2. **Model Optimization:** Addestramento di reti YOLO allo stato dell'arte con tecniche di *Hard Negative Mining* e *Test-Time Augmentation* (TTA) iniettata tramite *Monkey Patching*, innalzando nettamente l'F1-Score sui task più complessi (es. scritte pubblicitarie).

3. **Software Architecture:** Implementazione di un *Inference Server* disaccoppiato basato su MQTT, abbattendo l'overhead di rete derivato dal trasferimento di immagini Base64.

Il sistema risponde pienamente alle esigenze aziendali di TimeLapseLab, automatizzando la messa in sicurezza dei dati sensibili sui cantieri.

## 6.4 Sviluppi Futuri

Sebbene il sistema attuale sia già pronto per il *deployment*, esistono ampi margini per ulteriori ottimizzazioni architetturali:

- **Accelerazione Hardware e Quantizzazione:** La migrazione dei modelli esportati dal formato PyTorch/ONNX verso **TensorRT** (il motore di inferenza ad alte prestazioni di NVIDIA) permetterà di dimezzare la precisione dei pesi (da FP32 a FP16 o INT8), riducendo drasticamente i tempi di inferenza su server equipaggiati con GPU aziendali.

- **Dynamic Tiling:** Attualmente i parametri di *slice* di SAHI sono statici. Sviluppare un algoritmo di *Dynamic Tiling* che adatti le dimensioni della finestra di scorrimento in base alla densità di oggetti rilevata in una pre-scansione a bassa risoluzione potrebbe ridurre le inferenze inutili su vaste aree di background (ad esempio il cielo).

- **Edge Computing:** Traslare parte dell'inferenza direttamente a bordo delle telecamere di cantiere, riducendo i costi di banda per il trasferimento delle immagini verso il server centrale.

# Appendice A

# Codice Sorgente

In questa appendice vengono riportati gli script integrali sviluppati durante l'attività di tirocinio, citati nei capitoli precedenti. I codici includono la logica di preprocessing geometrico, la gestione asincrona del server HTTP e il *Worker* indipendente basato su MQTT, oltre che gli script di training e di testing.

## A.1 Script per Detection

```
# DETECTOR

import json
import torch
from pathlib import Path
from typing import List, Dict, Any, Union

from sahi import AutoDetectionModel
from sahi.predict import get_sliced_prediction

SLICE_HEIGHT = 800
SLICE_WIDTH =  1400
OVERLAP_RATIO = 0.15

class YoloTTAWrapper:
    def __init__(self, model):
        self.model = model

    def __call__(self, *args, **kwargs):
        kwargs['augment'] = True  # Test-Time Augmentation Forzata
        return self.model(*args, **kwargs)

    def __getattr__(self, item):
        return getattr(self.model, item)

```

```
26
27 class MultiModelDetector:
28     def __init__(self, models_config: Dict[str, str], use_tta: bool =
    True):
29         """
30         models_config: Un dizionario che mappa il nome della classe al
    percorso del modello.
31         Esempio:
32         {
33             'plate': 'models/best_plate.pt',
34             'person': 'models/yolov8_face.pt',
35             'truck': 'models/yolov8_vehicles.pt'
36         }
37         """
38         self.models = {}
39         self.device = "cuda:0" if torch.cuda.is_available() else "cpu"
40         print(f"Il detector utilizza il dispositivo: {self.device}")
41         for class_name, model_path in models_config.items():
42             print(f"Caricamento modello per {class_name}...")
43             model_kwargs = {}
44             try:
45                 sahi_model = AutoDetectionModel.from_pretrained(
46                     model_type='yolov11',
47                     model_path=model_path,
48                     confidence_threshold=0.3,
49                     device=self.device)
50                 if use_tta:
51                     sahi_model.model = YoloTTAWrapper(sahi_model.model)
52                     print(f"TTA forzata e abilitata per il modello {
    class_name}")
53                 self.models[class_name] = sahi_model
54                 print(f"Modello {class_name} caricato")
55             except Exception as e:
56                 print(f"Errore caricamento modello {class_name}: {e}")
57
58     def detect(self, image_array: Any, conf_dict: Dict[str, float] = None
    ) -> List[Dict[str, Any]]:
59         """
60         Esegue l'inferenza usando il modello specifico per ogni classe
    richiesta in conf_dict.
61         """
62         if not conf_dict:
63             return []
64
65         all_detections = []
66
67         # Itero su ogni classe richiesta dal worker
```

```
        for class_to_detect, threshold in conf_dict.items():

            # 1. Recupero modello
            model = self.models.get(class_to_detect)

            if not model:
                print(f"Nessun modello trovato per la classe {class_to_detect}")
                continue

            # 2. Aggiornamento soglia modello
            model.confidence_threshold = threshold

            # 3. Esecuzione SAHI
            result = get_sliced_prediction(
                image_array,
                model,
                slice_height=SLICE_HEIGHT,
                slice_width=SLICE_WIDTH,
                overlap_height_ratio=OVERLAP_RATIO,
                overlap_width_ratio=OVERLAP_RATIO,
                verbose=0
            )

            # 4. Estrazione risultati
            for object_prediction in result.object_prediction_list:
                pred_class_name = object_prediction.category.name.lower()
                confidence = float(object_prediction.score.value)
                if confidence < threshold:
                    continue
                bbox = object_prediction.bbox
                x1, y1, x2, y2 = int(bbox.minx), int(bbox.miny), int(bbox.maxx), int(bbox.maxy)
                w = x2 - x1
                h = y2 - y1

                # --- FILTRO GEOMETRICO ---
                if h == 0 or w == 0: continue
                aspect_ratio = w / h

                detection = {
                    "class": pred_class_name,
                    "confidence": round(confidence, 2),
                    "box": {"x": x1, "y": y1, "w": w, "h": h},
                    "coords_xyxy": [x1, y1, x2, y2]
                }
                all_detections.append(detection)
```

```
113
114         return all_detections
115
116     def save_detections_to_json(self, detections: List[Dict], output_path
    : Union[str, Path], metadata: Dict = None):
117         """
118         Salva i risultati dell'inferenza in un file JSON.
119         """
120         data_to_save = {
121             "detections": detections
122         }
123
124         if metadata:
125             data_to_save.update(metadata)
126
127         path_obj = Path(output_path)
128         path_obj.parent.mkdir(parents=True, exist_ok=True)
129
130         try:
131             with open(path_obj, 'w', encoding='utf-8') as f:
132                 json.dump(data_to_save, f, indent=4)
133             print(f"JSON salvato in: {output_path}")
134         except Exception as e:
135             print(f"Errore salvataggio JSON: {e}")
```

Listing A.1: Script che definisce la classe utilizzata per applicare la detection sfruttando i vari modelli.

## A.2 Script per Offuscamento

```
1  import cv2
2  import numpy as np
3  from typing import List, Dict
4
5
6  class ImageObfuscator:
7      def __init__(self):
8          pass
9
10     def obfuscate(self, image: np.ndarray, detections: List[Dict],
11                   blur_configs: Dict) -> np.ndarray:
12         """
13         Applica l'oscuramento basandosi sulle configurazioni del database
       per ogni classe.
14         """
```

```
        if not detections:
            return image

        h_img, w_img = image.shape[:2]

        for det in detections:
            x1, y1, x2, y2 = det['coords_xyxy']
            classe_rilevata = det['class']

            # 1. Recupero parametri blur per la classe
            cfg = blur_configs.get(classe_rilevata, {})

            forma = cfg.get('forma_maschera', 'rettangolare').lower()
            tipo = cfg.get('tipo_maschera', 'blur').lower()
            k_size = int(cfg.get('kernel_size', 61))
            gray_val = int(cfg.get('paint_grayscale_value', 125))
            multiplier = float(cfg.get('area_multiplier', 1.0))

            if k_size % 2 == 0:
                k_size += 1

            # 2. Area multiplier
            w = x2 - x1
            h = y2 - y1
            cx = x1 + w // 2
            cy = y1 + h // 2

            new_w = int(w * multiplier)
            new_h = int(h * multiplier)

            # Ricalcolo coordinate assicurandomi di non uscire dall'
    immagine
            nx1 = max(0, cx - new_w // 2)
            ny1 = max(0, cy - new_h // 2)
            nx2 = min(w_img, cx + new_w // 2)
            ny2 = min(h_img, cy + new_h // 2)

            roi = image[ny1:ny2, nx1:nx2]
            w_final = nx2 - nx1
            h_final = ny2 - ny1

            if w_final <= 0 or h_final <= 0:
                continue

            # 3. Tipo di maschera
            if tipo == 'paint':
```

```
                processed_roi = np.full((h_final, w_final, 3), gray_val, dtype=np.uint8)
            else:
                processed_roi = cv2.GaussianBlur(roi, (k_size, k_size), gray_val)

            # 4. Forma maschera
            if forma == 'ellittica':
                mask = np.zeros((h_final, w_final, 3), dtype=np.uint8)
                center = (w_final // 2, h_final // 2)
                axes = (w_final // 2, h_final // 2)
                cv2.ellipse(mask, center, axes, 0, 0, 360, 255, -1)

                # Applicazione ellisse
                mask_3ch = cv2.merge([mask, mask, mask])
                processed_roi = np.where(mask_3ch > 0, processed_roi, roi)

            # 5. Sovrascrittura immagine
            image[ny1:ny2, nx1:nx2] = processed_roi

        return image
```

Listing A.2: Script che definisce la classe utilizzata per applicare i filtri di anonimizzazione, sfruttando i parametri definiti lato server.

## A.3 Web Server Asincrono (HTTP e MQTT)

```
import logging
from aiohttp import web
from pathlib import Path
import base64
import asyncio
import paho.mqtt.client as paho
import signal
import sys

# CONFIGURAZIONE LOG
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s  [%(levelname)s]  %(message)s",
    force=True
)
logger = logging.getLogger(__name__)
client = paho.Client(client_id="http_server")
```

```
handlers = {}
#loop = asyncio.get_event_loop()
loop = None
MAX_SIZE_BODY = 10 * (1024 ** 2)

def on_message(client, userdata, msg):
    try:
        req_id = msg.topic
        fut = handlers.get(req_id, None)
        if fut and not fut.done():
            payload = msg.payload.decode()
            logger.info(f"Foto processata... Payload messaggio mqtt: {payload}")
            loop.call_soon_threadsafe(fut.set_result, True)
        else:
            logger.info(f"Fut non e' inizializzato (forse msg mqtt duplicato)... Payload messaggio mqtt: {msg.payload}")
    except Exception as e:
        logger.error(f"Errore in on_message: {e}")

def on_subscribe(client, userdata, mid, granted_qos):
    logger.info(f"\tIscrizione al topic {mid}")

def clientMqtt_configure():
    client.on_message = on_message
    client.on_subscribe = on_subscribe
    client.connect("localhost", 1883)
    client.loop_start()

async def handler_request(request):
    req_id = ""
    try:
        logger.info("===> Nuova richiesta in arrivo")
        type = request.match_info.get('type')
        nameFile = request.match_info.get('nameFile')
        logger.info(f"Endpoint chiamato: type={type}, nameFile={nameFile}")
        # Verifica corpo
        if request.content_length is None or request.content_length == 0:
            logger.warning("Nessun body fornito nella richiesta")
            return web.Response(status=400, text="Missing body")
        status = False
        try:
            #logger.info("Lettura JSON...")
            body = await request.json()
            logger.info("JSON letto correttamente")
        except Exception as e:
```

```
            logger.error(f"Errore parsing JSON: {e}")
            return web.Response(status=400, text="Invalid JSON")
        if "image" not in body or "id" not in body:
            logger.warning("Campo 'image' assente nel JSON")
            return web.Response(status=400, text="Body json doesn't have
    the correct field about image")
        try:
            #logger.info("Decodifica base64...")
            bytes_img = base64.b64decode(body['image'])
            logger.info("Decodifica base64 completata")
        except Exception as e:
            logger.error(f"Errore decodifica base64: {e}")
            return web.Response(status=400, text="Invalid base64 image")
        req_id = body['id']
        try:
            # logger.info("Scrittura file immagine...")
            image_path = Path("blurred_output/" + req_id)
            image_path.write_bytes(bytes_img)
            logger.info("Scrittura file completata")
        except Exception as e:
            logger.error(f"Errore scrittura file: {e}")
            return web.Response(status=500, text="File write error")

        # RICHIAMO FUNZIONE DI INFERENZA

        """Esempio di JSON:
{
    "id" : "image_name",
    "inferencePars" : "person=0.5, plate=0.6",
    "blurConfigs": {
        "plate": {
            "forma_maschera": "rettangolare",
            "tipo_maschera": "blur",
            "kernel_size": 61,
            "paint_grayscale_value": 125,
            "area_multiplier": 1.2
        },
        "person": {
            "forma_maschera": "ellittica",
            "tipo_maschera": "paint",
            "kernel_size": 31,
            "paint_grayscale_value": 200,
            "area_multiplier": 2.0
        }
    }
}
"""
```

```
        # 1. PREPARAZIONE PARAMETRI PER IL WORKER
        # Lettura parametri dal body HTTP
        raw_pars = body.get('inferencePars', "")
        blur_configs = body.get('blurConfigs', {})
        # Creazione payload completo per il worker
        job_payload = {
            "id": req_id,
            "camera_name": nameFile,
            "inference_pars": raw_pars,
            "blur_configs": blur_configs
        }
        # 2. PREPARAZIONE ASYNC (Future)

        # Creazione future
        fut = loop.create_future()
        handlers[req_id] = fut
        # Iscrizione al topic
        client.subscribe(req_id, qos=1)
        logger.info(f"Aspettando la conferma via mqtt")
        # 3. INVIO ORDINE AL WORKER
        import json
        logger.info(f"Delegando lavoro al worker per ID: {req_id}")
        # Publish su "Notifiche" di tutto il pacchetto dati
        client.publish("Notifiche", json.dumps(job_payload))
        # 4. ATTESA PASSIVA
        logger.info(f"In attesa del Worker...")
        try:
            # Blocco di questa richiesta finche' il future non e' set\
    _result
            # Aggiunta del timeout di 20 secondi
            result_json = await asyncio.wait_for(fut, timeout=20.0)
            # Risultati del worker in result\_json
            logger.info(f"Worker ha completato! Risposta: {result_json}")
            return web.Response(status=200, text=f"Response about {req_id
    } - inferenza eseguita con successo")
        except asyncio.TimeoutError:
            logger.warning(f"Timeout inferenza scaduto per {req_id}")
            return web.Response(status=504, text=f"Inferenza fallita -
    Response about {req_id}")

    # Catch generico a fine handler
    except Exception as e:
        logger.error(f"Errore inatteso del server: {e}")
        return web.Response(status=500, text="Internal Server Error")
    finally:
        handlers.pop(req_id, None)
        client.unsubscribe(req_id)
```

```
152
153 app = web.Application(client_max_size=MAX_SIZE_BODY)
154 app.add_routes([web.post('/{type}/{nameFile}', handler_request)])
155
156 async def shutdown(runner):
157     logger.info("Shutting down...")
158     # STOP MQTT
159     try:
160         client.loop_stop()
161         client.disconnect()
162     except:
163         pass
164     # STOP HTTP
165     await runner.cleanup()
166     logger.info("Bye")
167
168 async def main():
169     port = 8080
170     global loop
171     loop = asyncio.get_running_loop()
172     runner = web.AppRunner(app)
173     clientMqtt_configure()
174     await runner.setup()
175     site = web.TCPSite(runner, host="0.0.0.0", port=port)
176     await site.start()
177     logger.info(f"Server started on port: {port}!")
178     stop_event = asyncio.Event()
179     # Handle Ctrl-C
180     def _stop(*_):
181         logger.info("\tSIGINT received")
182         stop_event.set()
183     if sys.platform != 'win32':
184         loop.add_signal_handler(signal.SIGINT, _stop)
185         loop.add_signal_handler(signal.SIGTERM, _stop)
186     try:
187         await stop_event.wait()
188     except KeyboardInterrupt:
189         logger.info("\tSIGINT received (Windows)")
190     finally:
191         await shutdown(runner)
192
193 if __name__ == '__main__':
194     try:
195         asyncio.run(main())
196     except KeyboardInterrupt:
197         pass # Ignora l'errore a schermo quando premi Ctrl+C su Windows
```

Listing A.3: Implementazione del server aiohttp per la ricezione asincrona

## A.4 Inference Worker Indipendente

```
import time
import logging
import json
import cv2
import numpy as np
from pathlib import Path
import paho.mqtt.client as paho
import base64
# IMPORT
from detector import MultiModelDetector
from obfuscator import ImageObfuscator
# CONFIGURAZIONE
logging.basicConfig(level=logging.INFO, format="%(asctime)s [(WORKER)] %(message)s")
logger = logging.getLogger(__name__)

OUTPUT_DIR = Path("blurred_output")
JSON_DIR = Path("json_output")
JSON_DIR.mkdir(exist_ok=True)

MODELS_FOLDER = "models"
BROKER_ADDRESS = "localhost"
TOPIC_SUBSCRIBE = "Notifiche"
MODELS_MAP = {
    "plate": f"{MODELS_FOLDER}/modello_targhe.pt",
    "Hardhat": f"{MODELS_FOLDER}/modello_HH.pt",
}
# Inizializzazione Globale dei Modelli
detector = None
obfuscator = None
# GESTIONE MQTT
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        logger.info(f"Connesso al Broker MQTT. Iscrizione a '{TOPIC_SUBSCRIBE}'...")
        client.subscribe(TOPIC_SUBSCRIBE)
    else:
        logger.error(f"Errore connessione MQTT, codice: {rc}")

def on_message(client, userdata, msg):
```

```
    try:
        # Decodifica il payload JSON inviato dal server
        payload_str = msg.payload.decode()
        job_data = json.loads(payload_str)
        req_id = job_data['id']
        logger.info(f"Ricevuto job per ID: {req_id}")
        start = time.time()
        process_job(job_data, client)  # Passo tutto il dizionario dati
        end = time.time()
        total_time = end - start
        logger.info(f"Tempo totale per processare immagine: {total_time}
    s")
    except Exception as e:
        logger.error(f"Errore lettura messaggio MQTT: {e}")

def process_job(job_data, mqtt_client):
    req_id = job_data['id']
    # base64_image = job_data['decoded_image']
    camera_name = job_data['camera_name']
    blur_configs = job_data['blur_configs']
    raw_pars = job_data['inference_pars']
    # Parsing della confidenza dai parametri ricevuti
    conf_dict = {}
    if raw_pars:
        try:
            conf_dict = {
                k.strip().lower(): float(v.strip())
                for k, v in (item.split('=') for item in raw_pars.split('
    ,') if '=' in item)
            }
        except Exception as e:
            logger.warning(f"Errore nel parsing di inferencePars: {e}")
    try:
        # 1. CARICAMENTO: Decodifica base64 -> Bytes -> NumPy Array ->
    Immagine OpenCV
        """img_bytes = base64.b64decode(base64_image)
        np_arr = np.frombuffer(img_bytes, np.uint8)
        img = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)"""
        image_path = OUTPUT_DIR / req_id
        img_bgr = cv2.imread(str(image_path))
        if img_bgr is None:
            logger.error(f"Impossibile decodificare l'immagine base64 per
     l'ID: {req_id}")
            mqtt_client.publish(req_id, json.dumps({"status": "ERROR"}))
            return
        img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

```

```
        # 2. INFERENZA
        detections = detector.detect(img_rgb, conf_dict=conf_dict)
        logger.info(f"Applicata inferenza per immagine {req_id} -> {len(
    detections)} targhe trovate")

        # 3. SALVATAGGIO JSON
        json_path = str(JSON_DIR / f"{req_id}.json")
        metadata = {
            "id": req_id,
            "camera": camera_name,
            "timestamp": time.time(),
            "worker": "worker_01"
        }
        detector.save_detections_to_json(detections, json_path, metadata)
        logger.info(f"Salvate detection in json per immagine {req_id}")

        # 4. OFFUSCAMENTO
        img_blurred = obfuscator.obfuscate(img_bgr, detections,
    blur_configs)
        logger.info(f"Offuscata immagine {req_id}")

        # 5. SOVRASCRITTURA
        cv2.imwrite(str(image_path), img_blurred)
        logger.info("Salvata immagine offuscata")

        # 6. RISPOSTA AL SERVER
        # Pubblico sul topic ID per sbloccare il server che e' in 'await'
        response_msg = json.dumps({"status": "OK", "objects": len(
    detections)})
        mqtt_client.publish(req_id, response_msg)
        logger.info(f"Finito {req_id}. Risposto al server.")

    except Exception as e:
        logger.error(f"Errore worker: {e}")
        mqtt_client.publish(req_id, json.dumps({"status": "ERROR"}))

# MAIN LOOP
if __name__ == "__main__":
    # Caricamento modelli
    logger.info("Inizializzazione modelli...")
    detector = MultiModelDetector(MODELS_MAP)
    obfuscator = ImageObfuscator()

    # Configuro MQTT
    client = paho.Client(client_id="python_inference_worker")
    client.on_connect = on_connect
    client.on_message = on_message
```

```
126
127     logger.info(f"Connessione al broker {BROKER_ADDRESS}...")
128     try:
129         client.connect(BROKER_ADDRESS, 1883, 60)
130         # Loop infinito che ascolta i messaggi
131         client.loop_forever()
132     except Exception as e:
133         logger.error(f"Impossibile connettersi a MQTT: {e}")
```

Listing A.4: Worker MQTT per l'inferenza YOLO e l'offuscamento

## A.5 Script per il training su GPU Vast.ai

```
1  # ==========================================
2  # 1. INSTALLAZIONE LIBRERIA
3  # ==========================================
4  !pip install gdown
5
6  import gdown
7  import os
8
9  # ==========================================
10 # 2. DOWNLOAD DAL DRIVE
11 # ==========================================
12 FILE_ID = 'ID_DRIVE'
13
14 # Costruzione dell'URL diretto per il download
15 url = f'https://drive.google.com/uc?id={FILE_ID}'
16
17 # Definiamo dove salvare il file zip (dentro /workspace!)
18 percorso_zip = '/workspace/dataset_caschetti.zip'
19
20 print("Inizio il download da Google Drive...")
21 # Scaricamento del file mostrando la barra di progresso
22 gdown.download(url, percorso_zip, quiet=False)
23 print("Download completato!")
24
25 # ==========================================
26 # 3. ESTRAZIONE
27 # ==========================================
28 # La cartella di destinazione
29 cartella_destinazione = '/workspace/dataset_caschetti_totale'
30 os.makedirs(cartella_destinazione, exist_ok=True)
31
32 print("\nInizio l'estrazione delle immagini... (potrebbe volerci qualche
      istante)")
```

```
!unzip -q {percorso_zip} -d {cartella_destinazione}

print(f"Estrazione completata! I tuoi dati sono pronti in: {
    cartella_destinazione}")
os.remove(percorso_zip)
print("File .zip eliminato per liberare spazio.")
# ==========================================
# 1. INSTALLAZIONE DI YOLO
# ==========================================
!pip install ultralytics

from ultralytics import YOLO

# ==========================================
# 2. CONFIGURAZIONE DEL MODELLO
# ==========================================
# Caricamento dell'architettura Medium
model = YOLO("yolo11m.pt")
project_path = '/workspace/Training_Runs'

print("Avvio dell'addestramento su GPU in corso...")

# ==========================================
# 3. TRAINING
# ==========================================
results = model.train(
    data="/workspace/dataset_caschetti_totale/data.yaml",
    epochs=50,
    imgsz=640,
    batch=32,
    project=project_path,
    name='train_caschetti_v2',
    save=True,
    device=0,
    workers=8
)

print("Addestramento completato con successo!")
```

Listing A.5: Script per importare pacchetti e dataset utili per il training da effettuare sulla GPU noleggiata su Vast.ai, e per effettuare il training completo sfruttando il calcolo parallelo fornito dalla GPU stessa.

## A.6 Script per il testing

```python
# TESTING CON METRICHE

import cv2
import os
from pathlib import Path
import csv
from tqdm import tqdm
# ========================================
# 1. CONFIGURAZIONE
# ========================================
DRIVE_DIR = "/content/drive/MyDrive/person_detection"
MODELS_DIR = "/content/drive/MyDrive/modelli_condivisi"
TEST_DIR = "/content/drive/MyDrive/person_detection/dataset_tll/test"

MODELS_TO_TEST = {
    "person": f"{DRIVE_DIR}/modello_persone.pt"
}

cd = {"person": 0.5}
IOU_THRESHOLD = 0.5

# Cartelle del dataset esportato da Roboflow
CARTELLA_BASE_TEST = f"{TEST_DIR}"
# LIVELLI_DIFFICOLTA = ["easy_test_images", "medium_test_images", "
    difficult_test_images"]
# CARTELLA_LABELS = "test/labels"
FILE_REPORT = "benchmark_completo.csv"
CLASS_MAP = {0: "person"}

blur_config = {
    "adv": {
        "forma_maschera": "rettangolare",
        "tipo_maschera": "blur",
        "kernel_size": 61,
        "paint_grayscale_value": 125,
        "area_multiplier": 1
    }
}


# ========================================
# 2. FUNZIONI DI SUPPORTO (Matematica)
# ========================================
def parse_yolo_label(txt_path, img_w, img_h):
    """Legge il file .txt di YOLO e converte le coordinate da
    normalizzate ad assolute."""
```

```
    boxes = []
    if not os.path.exists(txt_path):
        return boxes  # File non esiste = immagine vuota (Background)

    with open(txt_path, 'r') as f:
        for line in f.readlines():
            parts = line.strip().split()
            if len(parts) >= 5:
                class_id = int(parts[0])
                # YOLO format: class x_center y_center width height (tutti tra 0 e 1)
                cx, cy, w, h = map(float, parts[1:5])

                # Conversione in pixel assoluti (x1, y1, x2, y2)
                abs_w = int(w * img_w)
                abs_h = int(h * img_h)
                x1 = int((cx * img_w) - (abs_w / 2))
                y1 = int((cy * img_h) - (abs_h / 2))
                x2 = x1 + abs_w
                y2 = y1 + abs_h
                class_name = CLASS_MAP.get(class_id, "sconosciuto")

                boxes.append({
                    "class": class_name,
                    "coords": [x1, y1, x2, y2]
                })
    return boxes


def calcola_iou(boxA, boxB):
    """Calcola l'Intersection over Union (IoU) tra due rettangoli."""
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])

    interArea = max(0, xB - xA) * max(0, yB - yA)
    boxAArea = (boxA[2] - boxA[0]) * (boxA[3] - boxA[1])
    boxBArea = (boxB[2] - boxB[0]) * (boxB[3] - boxB[1])

    if float(boxAArea + boxBArea - interArea) == 0: return 0.0
    return interArea / float(boxAArea + boxBArea - interArea)

# ==========================================
# 3. MOTORE DI BENCHMARK
# ==========================================
risultati_finali = []
```

```
obfuscator = ImageObfuscator()

for nome_test, percorso_modello in MODELS_TO_TEST.items():
    print("\n" + "=" * 50)
    print(f"VALUTAZIONE MODELLO: {nome_test}")
    print("=" * 50)

    try:
        detector = MultiModelDetector({nome_test: percorso_modello})
    except Exception as e:
        print(f"Errore caricamento {nome_test}: {e}")
        continue

    # Contatori globali per il modello
    TP= 0
    FP= 0
    FN= 0
    cartella_immagini = os.path.join(CARTELLA_BASE_TEST, "images")
    cartella_labels = os.path.join(CARTELLA_BASE_TEST, "labels")
    cartella_output = os.path.join(CARTELLA_BASE_TEST, "output_oscurato")
    if not os.path.exists(cartella_immagini):
        print(f"Saltata cartella {cartella_immagini} (non trovata)")
        continue
    os.makedirs(cartella_output, exist_ok=True)

    print(f"\nAnalisi cartella: {cartella_immagini.upper()}")
    tot_immagini = os.listdir(cartella_immagini)

    for nome_file in tqdm(tot_immagini, desc= "Inferenza"):
        if not nome_file.lower().endswith(('.png', '.jpg', '.jpeg')):
    continue

        percorso_immagine = os.path.join(cartella_immagini, nome_file)
        img_bgr = cv2.imread(percorso_immagine)
        if img_bgr is None: continue
        img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

        img_h, img_w = img_bgr.shape[:2]

        # 1. Ground Truth
        nome_txt = os.path.splitext(nome_file)[0] + ".txt"
        percorso_txt = os.path.join(cartella_labels, nome_txt)
        ground_truths = parse_yolo_label(percorso_txt, img_w, img_h)
        # 2. Inferenza con SAHI
        predizioni_raw = detector.detect(img_rgb, conf_dict=cd)
        predictions = [p["coords_xyxy"] for p in predizioni_raw]

```

```
        # 3. Oscuramento Privacy e Salvataggio (VISUAL CHECK!)
        img_oscurata = obfuscator.obfuscate(img_bgr.copy(),
    predizioni_raw, blur_config)
        cv2.imwrite(os.path.join(cartella_output, nome_file),
    img_oscurata)

        # 4. Calcolo Metriche
        tp_img, fp_img = 0, 0
        gt_matched = [False] * len(ground_truths)

        # Teniamo traccia di quali ground_truth abbiamo gia' "coperto"
    con una predizione
        gt_matched = [False] * len(ground_truths)

        for pred in predizioni_raw:
            pred_box = pred["coords_xyxy"]
            pred_class = pred["class"]

            best_iou, best_gt_idx = 0, -1

            for idx, gt in enumerate(ground_truths):
                if gt_matched[idx]: continue
                # Controllo fondamentale: le classi devono combaciare!
                if gt["class"] != pred_class: continue

                iou = calcola_iou(pred_box, gt["coords"])
                if iou > best_iou:
                    best_iou, best_gt_idx = iou, idx

            if best_iou >= IOU_THRESHOLD:
                tp_img += 1
                gt_matched[best_gt_idx] = True
            else:
                fp_img += 1

        fn_img = gt_matched.count(False)

        TP += tp_img
        FP += fp_img
        FN += fn_img

    # Metriche per livello di difficolta'
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0.0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0.0
    f1_score = 2 * (precision * recall) / (precision + recall) if (
    precision + recall) > 0 else 0.0

```

```
    print(f"   -> TP: {TP} | FP: {FP} | FN: {FN}")
    print(f"   -> F1-Score:  {f1_score:.2%}")

    risultati_finali.append({
        "TP (Corretti)": TP,
        "FP (Falsi Positivi)": FP,
        "FN (Mancati)": FN,
        "Precision": round(precision, 4),
        "Recall": round(recall, 4),
        "F1_Score": round(f1_score, 4)
    })



# ==========================================
# 5. SALVATAGGIO REPORT
# ==========================================

with open(FILE_REPORT, mode='w', newline='', encoding='utf-8') as
    file_csv:
    campi = [ "TP (Corretti)", "FP (Falsi Positivi)", "FN (Mancati)", "
    Precision", "Recall", "F1_Score"]
    writer = csv.DictWriter(file_csv, fieldnames=campi)
    writer.writeheader()
    writer.writerows(risultati_finali)

print(f"\nTEST CONCLUSO! Le immagini oscurate sono nelle cartelle '
    output_oscurato'. Report in: {FILE_REPORT}")
```

Listing A.6: Script per testare i modelli ottenuti. Il programma permette di calcolare l'IoU, poi tramite la definizione di una soglia minima calcola veri positivi, falsi positivi e falsi negativi, ed infine tramite essi calcola le metriche definitive (precision, recall ed F1 score).

# Bibliografia


[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788.

[2] Ultralytics, *YOLOv11 Documentation and Repository*, GitHub, 2024. [Online]. Disponibile qui

[3] Medium, *YOLOv11 Architecture Explained: Next-Level Object Detection with Enhanced Speed and Accuracy.* Disponibile qui

[4] F. C. Akyon, S. O. Altinuc, and A. Temizel, *Slicing Aided Hyper Inference and Fine-Tuning for Small Object Detection*, in IEEE International Conference on Image Processing (ICIP), 2022.

[5] F. C. Akyon on Medium, *SAHI: A vision library for large-scale object detection & instance segmentation.* Disponibile qui

[6] ISO/IEC 20922:2016, *Information technology — Message Queuing Telemetry Transport (MQTT) v3.1.1*, International Organization for Standardization, 2016.

[7] Unione Europea, *Regolamento (UE) 2016/679 del Parlamento europeo e del Consiglio del 27 aprile 2016 relativo alla protezione delle persone fisiche con riguardo al trattamento dei dati personali (GDPR)*, Gazzetta Ufficiale dell'Unione Europea, L 119/1, 4 maggio 2016.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.